Apple II

Apple Pascal

Language Reference Manual

apple computer inc.

10260 Bandley Drive
Cupertino, California 95014
(408) 996-1010

030-0101-00

NOTICE

Apple Computer Inc. reserves the right to make improvements in the product described in this manual at any time and without notice.

DISCLAIMER OF ALL WARRANTIES AND LIABILITY

APPLE COMPUTER INC. MAKES NO WARRANTIES, EITHER EXPRESS OR IMPLIED, WITH RESPECT TO THIS MANUAL OR WITH RESPECT TO THE SOFTWARE DESCRIBED IN THIS MANUAL, ITS QUALITY, PERFORMANCE, MERCHANTABILITY, OR FITNESS FOR ANY PARTICULAR PURPOSE. APPLE COMPUTER INC. SOFTWARE IS SOLD OR LICENSED "AS IS". THE ENTIRE RISK AS TO ITS QUALITY AND PERFORMANCE IS WITH THE BUYER. SHOULD THE PROGRAMS PROVE DEFECTIVE FOLLOWING THEIR PURCHASE, THE BUYER (AND NOT APPLE COMPUTER INC., ITS DISTRIBUTOR, OR ITS RETAILER) ASSUMES THE ENTIRE COST OF ALL NECESSARY SERVICING, REPAIR, OR CORRECTION AND ANY INCIDENTAL OR CONSEQUENTIAL DAMAGES. IN NO EVENT WILL APPLE COMPUTER INC. BE LIABLE FOR DIRECT, INDIRECT, INCIDENTAL, OR CONSEQUENTIAL DAMAGES RESULTING FROM ANY DEFECT IN THE SOFTWARE, EVEN IF APPLE COMPUTER INC. HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. SOME STATES DO NOT ALLOW THE EXCLUSION OR LIMITATION OF IMPLIED WARRANTIES OR LIABILITY FOR INCIDENTAL OR CONSEQUENTIAL DAMAGES, SO THE ABOVE LIMITATION OR EXCLUSION MAY NOT APPLY TO YOU.

1026 Bandley Drive
Cupertino, California 95014
(408) 996-1010

APPLE Product #A2L0027
(030-0101-00)

Apple II

Apple Pascal

Language Reference Manual

ACKNOWLEDGEMENTS

The Apple Pascal™ system incorporates UCSD Pascal™ and Apple extensions for graphics and other functions. UCSD Pascal was developed largely by the Institute for Information Science at the University of California at San Diego, under the direction of Kenneth L. Bowles.

TABLE OF CONTENTS

CHAPTER 3

BUILT-IN PROCEDURES AND FUNCTIONS

CHAPTER 4

THE PASCAL COMPILER 57

CHAPTER 5

PROGRAMS IN PIECES 71

CHAPTER 6

OTHER DIFFERENCES 83

CHAPTER 7

SPECIAL UNITS FOR THE APPLE 89

APPENDIX A

DEMONSTRATION PROGRAMS 107

APPENDIX B

TABLES 131

APPENDIX C

ADDITIONAL TEXT I/O DETAILS 143

APPENDIX D

ONE-DRIVE STARTUP 147

APPENDIX E

TWO-DRIVE STARTUP 169

APPENDIX F

APPLE PASCAL SYNTAX 191

INDEX 205

CHAPTER 1
INTRODUCTION

GETTING STARTED

If you don't already know how to start up the Apple Pascal Operating System for use with the Apple Pascal language, please read Appendix D if you have one diskette drive, or Appendix E if you have two or more diskette drives. Each of these Appendices is a tutorial session, covering system startup, diskette initialization, diskette copying, and a demonstration of Apple Pascal programming.

SCOPE OF THIS DOCUMENT

This document covers the features of the Apple Pascal programming language that are different from the "Standard Pascal" language defined by Jensen and Wirth in the Pascal User Manual and Report (Springer-Verlag, New York, 1978). This includes the differences introduced in UCSD Pascal, and also special extensions of UCSD Pascal for the Apple computer.

The Apple Pascal system facilities such as the Editor, the Linker, etc. are covered in the Apple Pascal Operating System Reference Manual. These facilities are useful in various applications besides Apple Pascal programming; they are discussed here only as they relate specifically to Apple Pascal programs.

HOW TO USE THIS DOCUMENT

To use this document you must either have a thorough knowledge of Standard or UCSD Pascal, or use some book or manual that fully describes Standard or UCSD Pascal. This is a reference manual, designed to give you the facts without very much emphasis on teaching you Pascal.

You should also have the Apple Pascal Operating System Reference Manual, which gives complete information on the various system facilities that support the creation and development of Apple Pascal programs.

One aspect of the Apple Pascal Operating System is covered in this manual: the procedures for starting up the system when your purpose is to work with Apple Pascal programs. Appendices D and E describe these procedures.

At various places in the text you will see the special symbol

which indicates a feature that you need to be cautious about. Another special symbol is

which indicates a particularly useful piece of information (usually something that is not obvious).

ORGANIZATION

Chapters 2 and 3 cover the large differences in Apple Pascal that will have the most immediate programming impact: the differences in predefined types, procedures, and functions, especially the procedures for input and output.

Chapter 4 covers the compiler operation and the compiler options, which are powerful and important. Further details on compiler operation can be found in the Apple Pascal Operating System Reference Manual.

Chapter 5 covers techniques for breaking a program into separate pieces which can be linked together. These techniques are another major area of difference but are not needed for small programs.

Chapter 6 gives the remaining differences in the language, which are of minor impact for most programs.

Chapter 7 covers the extremely powerful library options of Apple Pascal, including the Turtlegraphics package.

Appendix A presents a fully annotated program that uses graphics, and also describes the demonstration programs supplied with Apple Pascal.

Appendix B contains various tables relating to the Apple Pascal Language and the system.

Appendix C gives some technical details on textfile I/O operations.

Appendices D and E cover system startup and essential operating procedures for use with the Apple Pascal language.

Appendix F is a complete set of syntax diagrams for the Apple Pascal language.

NOTATION USED IN THIS MANUAL

In syntax descriptions, the following convention is used:

- Square brackets [] are used to enclose anything that may legally be omitted from the syntax.

DIFFERENCES BETWEEN APPLE AND STANDARD PASCAL

The major differences are summarized below; see Chapter 6 for the minor ones.

PREDEFINED VARIABLE TYPES

- A new variable type, STRING, supported by a set of new built-in procedures and functions. See Chapters 2 and 3.
- A new file type, INTERACTIVE, supported by the extended file I/O procedures and functions. See Chapters 2 and 3.
- Minor restrictions on SET types.
- Minor differences in the treatment of PACKED variables. Automatic PACK and UNPACK operations, with elimination of the PACK and UNPACK procedures of Standard Pascal. See Chapter 2.
- An extension of the INTEGER type called LONG INTEGER. A LONG INTEGER is a value represented by up to 36 binary-coded decimal (BCD) digits. See Chapter 2.

BUILT-IN PROCEDURES AND FUNCTIONS

These are the procedures and functions that are part of the Apple Pascal language itself, as opposed to special-purpose functions implemented in the system library. Built-in procedures and functions are called "built-ins" for short.

- New built-ins supporting STRING variables. See Chapters 2 and 3.
- Extended definitions of the built-ins for file I/O, supporting INTERACTIVE files. See Chapters 2 and 3.
- A set of new byte-oriented built-ins. See Chapter 3.
- New built-ins called MARK and RELEASE which replace the DISPOSE of Standard Pascal. See Chapter 3.
- Other new built-ins and redefinitions of Standard Pascal built-ins. See Chapter 3.
- The transcendental functions SIN, COS, EXP, ATAN, LN, LOG, and SQRT are not built-ins in Apple Pascal. They are provided as library functions. See Chapter 7.

BREAKING PROGRAMS INTO PIECES

- SEGMENT procedures and functions, which reside in memory only when active. See Chapter 5.
- UNITS, which are separately compiled collections of procedures that can be integrated into any host program via a library facility. See Chapter 5.
- EXTERNAL procedures and functions, which are declared in an Apple Pascal program but implemented in assembly language and then integrated into a host program via the library facility. See Chapter 5.

SPECIAL UNITS FOR THE APPLE

- These are major facilities for the Apple, implemented as UNITs in a system library. They include the Turtlegraphics package for the high-resolution color display of the Apple. See Chapter 7.

CHAPTER 2
PREDEFINED TYPES

In addition to the predefined types of Standard Pascal (REAL, INTEGER, CHAR, ARRAY, etc.), Apple Pascal has a STRING type, an INTERACTIVE file type, and a LONG INTEGER type.

Also, the details of certain other predefined types differ from Standard Pascal.

THE STRING TYPE

Apple Pascal has a new predeclared type, STRING. The value of a STRING variable is a sequence of characters. Variables of type STRING are essentially PACKED ARRAYs OF CHAR that have a dynamically changing number of elements (characters). However, the value of a STRING variable cannot be assigned to a PACKED ARRAY OF CHAR, and the value of a PACKED ARRAY OF CHAR cannot be assigned to a STRING variable. Strings are supported by a set of built-in procedures and functions; see Chapter 3.

The number of characters in a string at any moment is the length of the string. The default maximum length of a STRING variable is 80 characters, but this can be overridden in the declaration of a STRING variable (up to the absolute limit of 255). To do so, put the desired maximum length in [brackets] after the type identifier STRING. Examples of declarations of STRING variables are:

```
TITLE: STRING;  (* defaults to a maximum length of 80 characters *)

NAME: STRING[30];  (* allows the STRING to be a maximum of 30
                      characters*)
```

The value of a STRING variable can be altered by using an assignment statement with a string constant or another STRING variable:

```
TITLE := '    THIS IS A TITLE    '

  or

NAME := TITLE
```

or by means of the READ procedure as described in the next chapter:

```
READLN(TITLE)
```

or by means of the STRING built-ins, also described in the next chapter:

```
NAME:= COPY(TITLE,1,30)
```

Note that a string constant may not contain an end-of-line; the constant must be on a single line in the program.

The individual characters within a STRING are indexed from 1 to the LENGTH of the STRING. LENGTH is a built-in function which is described in Chapter 3. For example, if TITLE is the name of a string, then

```
TITLE[1]
```

is a reference to the first character of TITLE, and

```
TITLE[ LENGTH(TITLE) ]
```

is a reference to the last character of TITLE.

A variable of type STRING may be compared to any other variable of type STRING or to a string constant, regardless of its current dynamic length. The comparison is lexicographical: i.e., one string is "greater than" another if it would come first in an alphabetic list of strings. The ordering of the ASCII character set (see Appendix B) is used to determine this. The following program is a demonstration of legal comparisons involving variables of type STRING:

```
PROGRAM  COMPARESTRINGS;
VAR S: STRING;
    T: STRING[40];

BEGIN
  S:= 'SOMETHING';
  T:= 'SOMETHING BIGGER';
  IF S = T THEN
    WRITELN('Strings do not work very well')
  ELSE
    IF S > T THEN
      WRITELN(S,' is greater than ',T)
    ELSE
      IF S < T THEN
        WRITELN(S,' is less than ',T);
  IF S = 'SOMETHING' THEN
    WRITELN(S,'  equals  ',S);
  IF S > 'SAMETHING' THEN
    WRITELN(S,' is greater than SAMETHING');
  IF S = 'SOMETHING                ' THEN
    WRITELN('BLANKS DON''T COUNT')
  ELSE
    WRITELN('BLANKS APPEAR TO MAKE A DIFFERENCE');
  S:='XXX';
  T:='ABCDEF';
  IF S > T THEN
    WRITELN(S,' is greater than ',T)
  ELSE
    WRITELN(S,' is less than ',T)
END.
```

The above program produces the following output:

```
SOMETHING is less than SOMETHING BIGGER
SOMETHING  equals  SOMETHING
SOMETHING is greater than SAMETHING
BLANKS APPEAR TO MAKE A DIFFERENCE
XXX is greater than ABCDEF
```

Strings can also be declared as constants, as in the following:

```
PROGRAM BAZ;

CONST SAMMY = 'Hi there, I''m Sammy the String!';

BEGIN
  WRITELN(SAMMY)
END.
```

The use of STRING variables is discussed further in the next chapter, in connection with the built-in procedures and functions of Apple Pascal.

A variable of type STRING cannot be indexed beyond its current dynamic length. The following sequence will result in an invalid index run-time error:

```
TITLE:= '1234';
TITLE[5]:= '5'
```

Beware of zero-length strings: they cannot be indexed at all without causing unpredictable results or a run-time error. If a program indexes a string that might have zero length, it should first use the LENGTH function to see if the length is zero. If the length is zero, the program should not execute statements that index the string. See Chapter 3 for details on the LENGTH function.

Notice that a string value containing only one character is not the same thing as a CHAR value; strings and CHARs are distinct data types. The one exception is that a string constant containing only one character has exactly the same form as a CHAR constant, and such a constant can be used as either a CHAR value or a string value.

You cannot define a function of type STRING. However, there are built-in functions of type STRING as described in the next chapter.

THE FILE TYPES

A NOTE ON TERMINOLOGY

For every file named F that is declared in a Pascal program, there is an automatically declared variable named F^. This is the "buffer variable" of the file. Some Pascal manuals also use the looser term "window" to describe the way that different file records can be loaded into the buffer variable. This manual, instead, talks about a "file pointer" associated with each open file. The file pointer points to one record in the file, which is called the "current record." Please understand that the file pointer is not a Pascal POINTER variable but just a convenient way of discussing file records.

The following sections describe Apple Pascal's special file features: the INTERACTIVE file type, untyped files, predefined files, and a special format for files of characters.

INTERACTIVE FILES

Like a TEXT file, an INTERACTIVE file is a file of characters. The difference is in the way INTERACTIVE and TEXT files are handled by the RESET, READ, and READLN procedures.

When a Pascal program READs characters from a TEXT file, the program must first open the file with RESET. RESET automatically performs a GET operation: that is, it loads the first character of the file into the file's buffer variable and then advances the file pointer to the next character. A subsequent READ or READLN with a variable of type CHAR begins its operation by first taking the character that is already in the buffer variable and then performing a GET.

If the file is of type INTERACTIVE instead of TEXT, the opening RESET does not perform a GET. The buffer variable is undefined and the file pointer points to the first character of the file instead of the second. Therefore, a subsequent READ or READLN has to begin its operation by first performing a GET and then taking the character that was placed in the buffer variable by the GET. This is the reverse of the READ sequence used with a TEXT file.

There is one primary reason for using the INTERACTIVE type. If a file is not a diskette file but represents a device such as the keyboard, it is not possible to perform a GET on it until a character has been typed. If RESET tried to do a GET, the program would then go no further until a character was typed. With the INTERACTIVE type, the program doesn't perform a GET until it is executing a READ or READLN. The standard predeclared files INPUT and OUTPUT are INTERACTIVE files representing the console keyboard and screen; another predefined file called KEYBOARD also represents the keyboard (see the section below on Predefined Files).

UNTYPED FILES

In addition to the standard file types and the INTERACTIVE type, Apple Pascal allows "untyped" files -- objects that are declared with the word FILE and nothing more. Example:

```
VAR F: FILE;
```

Untyped files can only be used with the built-in functions BLOCKREAD and BLOCKWRITE for high-speed data transfers.

An untyped file F can be thought of as a file without a buffer variable F^. All I/O to this file must be accomplished by BLOCKREAD and BLOCKWRITE. These functions are described in the next chapter.

PREDEFINED FILES

The standard predefined files INPUT and OUTPUT refer to the keyboard and the screen respectively. In addition to these, Apple Pascal provides a predefined file called KEYBOARD. The difference between INPUT and KEYBOARD is that when INPUT is used to refer to the keyboard, the typed characters are automatically displayed on the screen; when KEYBOARD is used, the characters are not automatically displayed. This allows a Pascal program to have complete control over the response to characters typed by the user.

All three predefined files are of type INTERACTIVE, and all three are automatically opened via RESET when the Pascal program begins executing.

TEXTFILES

The Apple Pascal system provides that a TEXT or INTERACTIVE diskette file that is created with ".TEXT" as the last part of its title has a special internal format. Such files are called "textfiles" in this manual. Do not confuse textfiles with files that are of type TEXT or INTERACTIVE but do not have titles ending in ".TEXT".

All parts of the Pascal System that deal with files of characters (such as the editor) are designed to use the special textfile format; and if a textfile is accessed by a Pascal program, then the Pascal program will also use the special format. Therefore, the normal procedure is to use a title ending in ".TEXT" whenever you create a diskette file of the Pascal type TEXT or INTERACTIVE. The format of a textfile is as follows:

At the beginning of the file is a 1Ø24-byte header page, which contains information for the use of the text editor. This space is respected by all portions of the system. When a user Pascal program creates a textfile (via REWRITE), the system will automatically create the header. When a user Pascal program accesses an existing textfile (via RESET) the system skips the header. In other words, the header is invisible to a user Pascal program using REWRITE and RESET.

When a program uses BLOCKREAD and BLOCKWRITE to access files, the special textfile structure is <u>not</u> respected.

The system will transfer the header only on a diskette-to-diskette transfer, and will omit it on a transfer to a serial device (thus transfers from diskette to a printer or to the console will omit the header).

Following the header page, the text content itself appears in 1Ø24-byte text pages. Each text page is a sequence of lines, and the last line on a page is followed by enough null characters (ASCII ØØ) to fill out the 1Ø24 bytes. A line is defined as:

```
[DLE indent] [text] CR
```

where the brackets indicate that the DLE and the indent code may be absent and the text itself may be absent.

CR is the "Carriage Return" control character (ASCII 13), and may be absent at the end of the last line in the file. DLE is the "Data Link Escape" control character (ASCII 16). If present it is followed by a code indicating the indentation of the line. The code is 32 + the number of spaces to indent. Thus any leading spaces on a line are replaced by the DLE and the indent code.

The DLE and indent code and the nulls at the end of a text page are, like the header, invisible to a Pascal program. The DLE and indent are automatically translated to leading spaces, and vice versa.

The end of the file is marked by the ETX control character (ASCII 3).

THE SET TYPES

APPLE Pascal supports all of the Standard Pascal constructs for sets. Two limitations are imposed on sets:

- A set may not have more than 512 elements assigned to it.

- A set may not have any INTEGERs less than Ø or greater than 511 assigned to it.

A set of 512 elements will occupy 32 words of memory.

Comparisons and operations on sets are allowed only between sets whose individual elements are of the same type. For example, in the sample program below, the base type of the set S is the subrange type Ø..49, while the base type of the set R is the subrange type 1..1ØØ. The underlying type of both sets is the type INTEGER, so the comparisons and operations on the sets S and R in the following program are legal:

```
PROGRAM SETCOMPARE;
VAR S: SET OF Ø..49;
    R: SET OF 1..1ØØ;

BEGIN
  S:= [Ø,5,1Ø,15,2Ø,25,3Ø,35,4Ø,45];
  R:= [1Ø,2Ø,3Ø,4Ø,5Ø,6Ø,7Ø,8Ø,9Ø];
  IF S = R THEN
    WRITELN('... oops ...')
  ELSE
    WRITELN('sets work');
  S := S + R
END.
```

In the following example, the comparison I = J is not legal since the two sets are of two distinct underlying types.

```
PROGRAM ILLEGALSETS;
TYPE STUFF=(ZERO,ONE,TWO);
VAR I: SET OF STUFF;
    J: SET OF Ø..2;

BEGIN
  I:= [ZERO];
  J:= [1,2];
  IF I = J THEN ...      <<<< error here
END.
```


PACKED VARIABLES

PACK AND UNPACK

Apple Pascal does not require the Standard Pascal procedures PACK and UNPACK, and these procedures are not provided. If a variable is PACKED, all required packing and unpacking are done automatically on an element-by-element basis.

PACKED FILES

Apple Pascal does not support PACKED FILE types. A PACKED FILE can be declared, but the data in the file will not actually be packed.

PACKED ARRAYS

The Apple Pascal compiler supports PACKED ARRAYs as defined in Standard Pascal. For example, consider the following declarations:

```
A: ARRAY[Ø..9] OF CHAR;
B: PACKED ARRAY[Ø..9] OF CHAR;
```

The array A will occupy ten 16-bit words of memory, with each element of the array occupying one word. The PACKED ARRAY B on the other hand will occupy a total of only 5 words, since each 16-bit word contains two 8-bit characters. Each element of B is 8 bits long.

PACKED ARRAYs need not be restricted to arrays of type CHAR. For example:

```
C: PACKED ARRAY[Ø..1] OF Ø..3;
D: PACKED ARRAY[1..9] OF SET OF Ø..15;
D2: PACKED ARRAY[Ø..239,Ø..319] OF BOOLEAN;
```

Each element of the PACKED ARRAY C is only 2 bits long, since only 2 bits are needed to represent the values in the range Ø..3. Therefore C occupies only one 16-bit word of memory, and 12 of the bits in that word are unused. The PACKED ARRAY D is a 9-word array, since each element of D is a SET which can be represented in a minimum of 16 bits. Each element of a PACKED ARRAY OF BOOLEAN, such as D2 in the above example, occupies only one bit.

The details of exactly how variables are packed are unspecified. In most cases, the minimum space into which an array can be packed is one word (two eight-bit bytes). For example, consider

```
BITS: PACKED ARRAY[Ø..7] OF BOOLEAN;
```

This is an eight-element array where each element requires one bit, so you might expect it to occupy eight bits or one byte. In fact, it occupies one word or two bytes. Furthermore, the two-dimensional array

```
BATS: PACKED ARRAY[Ø..3] OF PACKED ARRAY[Ø..7] OF BOOLEAN;
```

or its equivalent

```
BATS: PACKED ARRAY[Ø..3,Ø..7] OF BOOLEAN;
```

consists of four arrays. Each of them, like the previous array, occupies one word. Therefore BATS occupies four words.

Note that a PACKED ARRAY OF CHAR always occupies one byte per character and a PACKED ARRAY OF Ø..255 always occupies one byte per element.

Also, packing never occurs across word boundaries. This means that if the type of element to be packed requires a number of bits which does not divide evenly into 16, there will be some unused bits in each of the words where the array is stored.

The following two declarations are NOT equivalent because of the way the Pascal Compiler is implemented:

```
E: PACKED ARRAY[Ø..9] OF ARRAY[Ø..3] OF CHAR;
F: PACKED ARRAY[Ø..9,Ø..3] OF CHAR;
```

In the declaration of E, the second occurrence of the reserved word ARRAY causes the packing option in the compiler to be turned off. E becomes an unpacked array of 4Ø words. On the other hand, the PACKED ARRAY F occupies only 2Ø words because the reserved word ARRAY occurs only once in the declaration. If E is declared as

```
E: PACKED ARRAY[Ø..9] OF PACKED ARRAY[Ø..3] OF CHAR;
```

or as

```
E: ARRAY[Ø..9] OF PACKED ARRAY[Ø..3] OF CHAR;
```

F and E will have identical configurations.

In declaring a PACKED ARRAY, the word PACKED is only meaningful before the last appearance of the word ARRAY in the declaration. When in doubt, a good rule of thumb for declaring a multidimensional PACKED ARRAY is to place the word PACKED before every appearance of the word ARRAY to ensure that the resultant array will be PACKED.

The array will only be packed if the type of each element of the array is scalar, subrange, or a set and each array element can be represented in 8 bits or fewer. For an array whose elements are sets, this means that the underlying type of the set must not contain more than 8 elements, and must not contain any integer greater than 255.

The following declaration will result in no packing whatsoever because the final type of the array cannot be represented in a field of 8 bits:

```
G: PACKED ARRAY[Ø..3] OF Ø..1ØØØ;
```

G will be an array which occupies four 16-bit words.

Note that a string constant may be assigned to a PACKED ARRAY OF CHAR (if it has exactly the same length), but not to an unpacked ARRAY OF CHAR. Likewise, comparisons between an ARRAY OF CHAR and a string constant are illegal.

Because of their different sizes, PACKED ARRAYs cannot be compared to ordinary unpacked ARRAYs.

A PACKED ARRAY OF CHAR may be printed out with a single write statement (exactly as if it were a string):

```
PROGRAM VERYSLICK;
VAR T: PACKED ARRAY[Ø..1Ø] OF CHAR;
BEGIN
  T:='HELLO THERE';
  WRITELN(T)
END.
```

PACKED RECORDS

The following RECORD declaration declares a RECORD with four fields. The entire RECORD occupies one 16-bit word as a result of declaring it to be a PACKED RECORD.

```
VAR R: PACKED RECORD
         I,J,K: Ø..31;
         B: BOOLEAN
       END;
```

The variables I, J, K each take up five bits in the word. The boolean variable B is allocated to the 16th bit of the same word.

In much the same manner that PACKED ARRAYs can be multidimensional PACKED ARRAYs, PACKED RECORDS may contain fields which themselves are PACKED RECORDS or PACKED ARRAYS. Again, slight differences in the way in which declarations are made will affect the degree of packing

achieved. For example, note that the following two declarations are not equivalent:

```
VAR A:PACKED RECORD                VAR B:PACKED RECORD
      C:INTEGER;                         C:INTEGER;
      F:PACKED RECORD                    F:RECORD
                R: CHAR;                     R:CHAR;
                K: BOOLEAN                   K:BOOLEAN
              END;                         END;
      H:PACKED ARRAY[Ø..3] OF CHAR       H:PACKED ARRAY[Ø..3] OF CHAR
    END;                               END;
```

As with PACKED ARRAYs, the word PACKED should appear with every occurrence of the word RECORD in order for the PACKED RECORD to retain its packed qualities throughout all fields of the RECORD. In the above example, only RECORD A has all of its fields packed into one word. In B, the F field is not packed and therefore occupies two 16-bit words. It is important to note that a packed or unpacked ARRAY or RECORD which is a field of a PACKED RECORD will always start at the beginning of the next word boundary. This means that in the case of A, even though the F field does not completely fill one word, the H field starts at the beginning of the next word boundary.

A case variant may be used as the last field of a PACKED RECORD, and the amount of space allocated to it will be the size of the largest variant among the various cases. The actual nature of the packing is beyond the scope of this document.

```
VAR K: PACKED RECORD
         B: BOOLEAN;
         CASE F: BOOLEAN OF
           TRUE: (Z:INTEGER);
           FALSE: (M: PACKED ARRAY[Ø..3] OF CHAR)
       END;
```

In the above example the B and F fields are stored in two bits of the first 16-bit word of the record. The remaining fourteen bits are not used. The size of the case variant field is always the size of the largest variant, so in the above example, the case variant field will occupy two words. Thus the entire PACKED RECORD will occupy three words.

USING PACKED VARIABLES AS PARAMETERS

No PACKED variable may be passed as a VAR (call-by-reference) parameter to a PROCEDURE or FUNCTION. Packed variables may, however, be passed as ordinary call-by-value parameters.

THE LONG INTEGER TYPE

In Apple Pascal, the predefined INTEGER type can be modified by a length attribute as in the following examples:

```
TYPE BIGNUM = INTEGER[12];
VAR FATS: INTEGER[25];
```

This defines BIGNUM as a type which can have any integer value requiring not more than 12 decimal digits. FATS can have any integer value requiring not more than 25 digits. The length attribute can be any unsigned INTEGER constant up to and including 36.

This is a new kind of type, which is called a LONG INTEGER in this manual. The LONG INTEGER is suitable for business, scientific or other applications which need extended number lengths with complete accuracy. A LONG INTEGER is represented internally as a binary-coded decimal (BCD) number; that is, each decimal digit of the value is represented in binary. This means that there can be no rounding errors in working with LONG INTEGER values.

LONG INTEGER constants are also allowed. Any integer constant whose value exceeds MAXINT is automatically a constant of the type LONG INTEGER.

The integer arithmetic operations (+, -, *, and DIV) can all be used with LONG INTEGER values. However, MOD cannot be used with LONG INTEGERs. In integer arithemetic, overflow occurs if any intermediate or final result requires more than 36 decimal digits. When a LONG INTEGER value is assigned to a LONG INTEGER variable, overflow occurs if the value requires more decimal digits than the defined length of the variable.

An INTEGER value can always be assigned to a LONG INTEGER variable; it is automatically converted to the appropriate length. However, a LONG INTEGER value can never be assigned to an INTEGER variable. If INTEGER and LONG INTEGER values are mixed in an expression, the INTEGER values are converted to LONG INTEGER and the result is a LONG INTEGER value. LONG INTEGERs and REALs are incompatible; they can never be mixed in an arithmetic expression or assigned to each other.

All of the standard relational operators may be used with mixed LONG INTEGER and INTEGER values.

The built-in procedure STR accepts a LONG INTEGER value as a parameter, and converts it to a string of decimal digits. The built-in function TRUNC accepts a LONG INTEGER value as a parameter, and returns the corresponding INTEGER value if the absolute value of the LONG INTEGER is less than or equal to MAXINT. These built-ins are described in the next chapter; they are the only built-ins which accept LONG INTEGER parameters.

An attempt to declare a LONG INTEGER in a parameter list will result in a syntax error. This restriction may be circumvented by defining a type which is a LONG INTEGER. For example:

```
TYPE LONG = INTEGER[18];
PROCEDURE BIGNUMBER(BANKACCT: LONG);
```

EXAMPLES:

```
VAR I: INTEGER;
    L: INTEGER[N]; {where N is an integer constant <= 36 }
    R: REAL;

I:= L {syntax error; the TRUNC function can be used to convert a
       LONG INTEGER to an INTEGER}
L:=-L {correct, if -L does not require more than 36 digits; the
       minus sign doesn't count as a digit}
L:= I {always correct}
L:= R {never accepted}
R:= L {never accepted}
```

The memory space allocated for a LONG INTEGER is always an integral number of words. Specifically, a variable of type INTEGER[n] occupies

```
(n + 3) DIV 4 +1
```

words.

Therefore, the actual limit on the value of a LONG INTEGER may exceed the number of decimal digits specified in its declaration. For example, a length of 5 through 8 occupies three words and can store values up to and including 99999999; a length of 9 through 12 occupies four words and can store values up through 999999999999; a length of 13 through 16 occupies five words and can store values up through 9999999999999999.

CHAPTER 3
BUILT-IN PROCEDURES AND FUNCTIONS

This chapter describes all the built-in procedures and functions of Apple Pascal that differ from Standard Pascal. This does not include the procedures and functions that are provided as library UNITs, e.g. the graphics procedures and functions. Chapter 7 covers the library UNITs provided with Apple Pascal.

Transcendental functions (e.g. the trig functions SIN, COS, etc.) are a special case. In Standard Pascal they are built-in functions, but in Apple Pascal they are in a library UNIT. The ATAN and LOG functions differ slightly from Standard Pascal, and they are described in this chapter. The other transcendentals differ only in that to use them your program must include a USES TRANSCEND statement as described in Chapter 7.

Since some of these built-in procedures and functions do no checking for range validity of parameters, they may easily cause unpredictable results. Those built-ins which are particularily dangerous are noted as such in their descriptions. Any necessary range or validity checks are your responsibility.

STRING BUILT-INS

In the following descriptions, a "string value" means a string variable, a quoted string, or any function or expression whose value is a string. Unless otherwise stated all parameters are called by value.

THE LENGTH FUNCTION

The LENGTH function returns the integer value of the length of a string. The form is

```
LENGTH (STRG)
```

where STRG is a string value. Example:

```
GEESTRING := '1234567';
WRITELN( (LENGTH(GEESTRING), ' ', LENGTH('')) 
```

This will print:

```
7 Ø
```


THE POS FUNCTION

The POS function returns an integer value. The form is

```
POS (SUBSTRG, STRG)
```

where both SUBSTRG and STRG are string values. The POS function scans STRG to find the first occurrence of SUBSTRG within STRG. POS returns the index within STRG of the first character in the matched pattern. If the pattern is not found, POS returns zero. Example:

```
STUFF := 'TAKE THE BOTTLE WITH A METAL CAP';
PATTERN := 'TAL';
WRITELN( POS(PATTERN, STUFF) )
```

This will print:

```
26
```

THE CONCAT FUNCTION

The CONCAT function returns a string value. The form is

```
CONCAT ( STRGs )
```

where STRGs means any number of string values separated by commas. This function returns a string which is the concatenation of all the strings passed to it. Example:

```
SHORTSTRING := 'THIS IS A STRING';
LONGSTRING  := 'THIS IS A VERY LONG STRING.';
LONGSTRING := CONCAT('START ', SHORTSTRING, '-', LONGSTRING);
WRITELN(LONGSTRING)
```

This will print:

```
START THIS IS A STRING-THIS IS A VERY LONG STRING.
```

THE COPY FUNCTION

The COPY function returns a string value. The form is

```
COPY (STRG, INDEX, COUNT)
```

where STRG is a string value and both INDEX and COUNT are integer values. This function returns a string containing COUNT characters copied from STRG starting at the INDEXth position in STRG. Example:

```
TL := 'KEEP SOMETHING HERE';
KEPT := COPY(TL, POS('S', TL), 9);
WRITELN(KEPT)
```

This will print:

```
SOMETHING
```

THE DELETE PROCEDURE

The DELETE procedure modifies the value of a string variable. The form is

```
DELETE (STRG, INDEX, COUNT)
```

Where STRG is a string variable called by reference and modified, and both INDEX and COUNT are integer values. This procedure removes COUNT characters from STRG starting at the INDEX specified. Example:

```
OVERSTUFFED := 'THIS STRING HAS FAR TOO MANY CHARACTERS IN IT.';
DELETE(OVERSTUFFED, POS('HAS', OVERSTUFFED)+3, 8);
WRITELN(OVERSTUFFED)
```

This will print:

```
THIS STRING HAS MANY CHARACTERS IN IT.
```


THE INSERT PROCEDURE

The INSERT procedure modifies the value of a string variable. The form is

```
INSERT (SUBSTRG, STRG, INDEX)
```

where SUBSTRG is a string value, STRG is a string variable called by reference, and INDEX is an integer value. This inserts SUBSTRG into STRG at the INDEXth position in STRG. Example:

```
ID := 'INSERTIONS';
MORE := ' DEMONSTRATE';
DELETE(MORE, LENGTH(MORE), 1);
INSERT(MORE, ID, POS('IO', ID));
WRITELN(ID)
```

This will print:

```
INSERT DEMONSTRATIONS
```

THE STR PROCEDURE

The STR procedure modifies the value of a string variable. The form is

```
PROCEDURE STR ( LONG , STRG )
```

where LONG is an integer value, and STRG is a string variable called by reference. LONG may be a LONG INTEGER.

This converts the value of LONG into a string. The resulting string is placed in STRG. See Chapter 2 for more about the use of LONG INTEGERs. Example:

```
INTLONG := 1Ø2Ø395Ø3;
STR(INTLONG, INTSTRING);
INSERT('.', INTSTRING, LENGTH(INTSTRING)-1);
WRITELN('$', INTSTRING)
```

This will print:

```
$1Ø2Ø395.Ø3
```

The following program segment will provide a suitable dollar and cent routine:

```
STR(L,S); INSERT('.',S,LENGTH(S)-1); WRITELN(S)
```

where L and S are appropriately declared.

INPUT AND OUTPUT BUILT-INS

OVERVIEW OF APPLE PASCAL I/O FACILITIES

This section deals with data transfers to and from all peripheral devices, including diskette drives, the screen, the keyboard, printers, etc. There are also certain "integral" devices such as the TTL game-control outputs and the built-in speaker, which are not considered as I/O devices; see Chapter 7. For complete information on Apple Pascal file types, see Chapter 2.

Apple Pascal I/O facilities can be thought of as existing at four different levels:

- Hardware-oriented I/O: the UNITREAD, UNITWRITE, and UNITCLEAR procedures are the lowest level of control. They allow a Pascal program to transfer a specified number of consecutive bytes between memory and a device. They are not controlled by filenames, directories, etc., but merely use device numbers and (for diskette drives) block numbers.

- Untyped file I/O: The BLOCKREAD and BLOCKWRITE functions provide I/O for untyped files (see Chapter 2). They make use of filenames and directories but consider a file to be merely a sequence of bytes -- not a sequence of records of a particular type.

- Typed file I/O: The GET, PUT, and SEEK procedures treat a file as a sequence of records. GET and PUT provide transfers between individual file records and the file's buffer variable, and SEEK moves the pointer to a specified record within the file. The EOF function provides an indication of when the end of the file has been reached.

- Text file I/O: The READ, READLN, WRITE, and WRITELN procedures provide transfers between a file of type TEXT or INTERACTIVE and program variables. The PAGE procedure writes a top-of-form control character into a textfile. The EOLN function provides an indication of when the end of a text line has been reached. This is the highest level of I/O control, with many sophisticated features.

As mentioned in Chapter 2, the INPUT, OUTPUT, and KEYBOARD files are predefined and need not be declared in a program. All other files must first be declared in the VAR section of a program, and must then be opened by means of RESET or REWRITE before they can be used in any way.

Opening a file is a means of associating the file's identifier (declared in the program) with its title (used by the operating system). If the file to be used does not already exist, open it with REWRITE; this causes the operating system to create a directory entry for the file. If REWRITE is used with the title of an existing file, the existing file is destroyed and a new directory entry is created. RESET is used to open an existing file and can also be used to move the file pointer back to the beginning of a file that is already open. A CLOSE procedure is also provided. It offers several options for the disposition of the file when the program is through using it.

If an I/O operation is unsuccessful, the operating system will normally terminate program execution. However, there is a compiler option to disable this feature. The IORESULT function allows the program itself to check on the status of the most recent I/O operation and take appropriate action.

THE REWRITE PROCEDURE

This procedure creates a new file and marks the file as open. As explained below, it can also be used to open an existing file. The form is

```
REWRITE ( FILEID , TITLE )
```

where FILEID is the identifier of a previously declared file, and TITLE is a string containing any legal file title.

If the device specified in the TITLE is not a diskette, then the file is opened for both input and output; If the TITLE indicates a diskette file, REWRITE creates a new file and opens it for input and output.

If the file is already open, an I/O error occurs (see IORESULT below). The file remains open.

An example showing the use of REWRITE in a program follows the description of GET and PUT below.

THE RESET PROCEDURE

This procedure opens an existing file for both reading and writing. There are two forms:

```
RESET ( FILEID , TITLE )
RESET ( FILEID )
```

where FILEID is the identifier of a previously declared file, and TITLE is a string containing any legal file title.

If a TITLE is used and the specified file is already open, an I/O error occurs (see IORESULT, below). The file's state remains unchanged. If the file does not exist, an I/O error occurs.

A RESET without the TITLE can only be used on an open file; the effect is simply to reposition the file pointer as if the file had just been opened.

If the file is not of type INTERACTIVE, RESET automatically performs a GET action -- that is, it loads the first record of the file into the file's buffer variable and advances the file pointer to the second record. If the file is INTERACTIVE, no GET is performed; the buffer variable's value is undefined and the file pointer points to the first record. (GET is described further on.)

Note that RESETting a non-INTERACTIVE file to an output-only device, such as PRINTER:, may cause a run-time error as a result of the automatic GET caused by the RESET.

When an existing file is opened with RESET and is then used for output, only the file records actually written to are affected.

An example showing the use of RESET in a program follows the description of GET and PUT below.

THE CLOSE PROCEDURE

This procedure closes a file which was previously opened with RESET or REWRITE. The form is

```
CLOSE ( FILEID [, OPTION] )
```

where FILEID is the identifier of a previously declared file, and OPTION may be any one of the following:

NORMAL -- a normal close is done, i.e. CLOSE simply sets the file state to closed. If the file was opened using REWRITE and is a disk file, it is deleted from the directory.

LOCK -- the file is made permanent in the directory if the file is on a disk and the file was opened with a REWRITE; otherwise a NORMAL close is done. If the TITLE matches an existing diskette file, the original contents of the file are lost.

PURGE -- if the file is a diskette file, it is deleted from the directory. In the special case of a diskette file that already exists and is opened with REWRITE, the original file remains in the directory, unchanged. If the file is not a diskette file, the associated unit will go off-line.

CRUNCH -- this is like LOCK except that it locks the end-of-file to the point of last access, i.e. everything after the last element accessed is thrown away. If the TITLE matches an existing diskette file, the original contents of the file are lost.

If the OPTION is omitted, the NORMAL close is performed.

All CLOSEs regardless of the option will mark the file closed and will make the file buffer variable FILEID^ undefined. CLOSE on a CLOSEd file causes no action.

An example showing the use of CLOSE in a program follows the description of GET and PUT below.

THE EOF FUNCTION

This function returns a BOOLEAN value to indicate whether the end of a specified file has been reached. When EOF is true, nothing more can be read from the file. The form is

```
EOF [ ( FILEID ) ]
```

If (FILEID) is not present, INPUT is assumed.

EOF is false immediately after the file is opened, and true on a closed file. Whenever EOF (FILEID) is true, FILEID^ is undefined.

After a GET, EOF is true if the GET attempted to access a record that is after the end of the file. After a PUT or WRITE, EOF is true if the file cannot be expanded to accommodate the PUT or WRITE (because of limited diskette space, for example).

For details on EOF after a READ or READLN operation, see the descriptions of READ and READLN further on in this chapter, and Appendix C.

When EOF becomes true during a READ or GET operation, the value of FILEID^ is not defined.

When keyboard input is being read (via the predefined files INPUT or KEYBOARD), EOF only becomes true when the end-of-file character is typed. The end-of-file character is ctrl-C (ASCII 3). EOF remains true until the file INPUT or KEYBOARD is RESET, and no more typed characters can be read until this is done.

An example showing the use of EOF in a program follows the description of GET and PUT below.

THE EOLN FUNCTION

EOLN is defined only for a file of type TEXT, FILE OF CHAR, or INTERACTIVE. This function returns a BOOLEAN value to indicate whether the pointer for a specified text file is at the end of a line. The form is

```
EOLN [ ( FILEID ) ]
```

If (FILEID) is not present, INPUT is assumed.

EOLN returns false immediately after the file is opened, and true on a closed file.

When a GET finds an end-of-line character (the CR character, ASCII 13) in the file, it sets EOLN to true. Instead of loading the end-of-line character into the file's buffer variable it loads a space (ASCII 32).

For the behavior of EOLN after a READ or READLN, see the descriptions of these statements further on.

THE GET AND PUT PROCEDURES

These procedures are used to read or write one logical record from or to a typed file. The forms are

```
GET ( FILEID )
PUT ( FILEID )
```

where FILEID is the identifier of a previously declared typed file. A typed file is any file for which a type is specified in the variable declaration, as opposed to untyped files (see Chapter 2).

GET (FILEID) advances the file pointer to the next record and moves the contents of this record into the file buffer variable FILEID^. The next GET or PUT with the same FILEID will access the next record in sequence.

PUT (FILEID) advances the file pointer to the next record and puts the contents of FILEID^ into this record. The next GET or PUT with the same FILEID will access the next record in sequence.

The actual physical disk access may not occur until the next time the physically associated block of the disk is no longer considered the current working block. The kinds of operation which tend to force the block to be written are: a SEEK to elsewhere in the file, a RESET, and CLOSE. Successive GETs or PUTs to the file will cause the physical I/O to happen when the block boundaries are crossed.

The following two example programs illustrate the use of REWRITE, RESET, CLOSE, EOF, GET, and PUT. The first program creates a new file of type REAL, with the title REAL.DATA, and puts ten REAL values into it. The values are supplied by the user.

To obtain the values, the program uses a WRITE to display a prompt on the screen and a READ to accept the value typed by the user. READ and WRITE are described in detail further on in this chapter.

```
PROGRAM MAKEFILE;

VAR F: FILE OF REAL;
    I: INTEGER;

BEGIN
(*Open with REWRITE since this is a new file.*)
   REWRITE(F, '*REALS.DATA');
(*Read 10 numbers and store them in the file.*)
   FOR I:=1 TO 10 DO BEGIN
(*Put a prompt on the screen.*)
     WRITE('-->');
(*Read a number from the keyboard.*)
     READ(F^);
(*Store the number in the file.*)
     PUT(F)
   END;
(*Close the file and lock it.*)
   CLOSE(F, LOCK)
END.
```

The second program reads values from the file created by the first program, and displays them on the screen.

```
PROGRAM READFILE;

VAR F: FILE OF REAL;

BEGIN
(*Open with RESET since we want to read the file*)
   RESET(F, '*REALS.DATA');
(*Read each number from the file and display them*)
   WHILE NOT EOF(F) DO BEGIN
(*Display the current number on the screen*)
     WRITELN(F^);
(*Advance to the next number*)
     GET(F)
   END;
(*Close the file*)
   CLOSE(F)
END.
```

Note that these programs offer no flexibility as to the title of the file. The example under READLN below shows how to let the user specify the title of the file to be used.

THE IORESULT FUNCTION

This function returns an integer value which reflects the status of the last completed I/O operation. The form is

```
IORESULT
```

The values returned by IORESULT are as follows (also see Table 2):

Ø	No error; normal I/O completion
1	Bad block on diskette (not used on Apple)
2	Bad device (volume) number
3	Illegal operation (e.g., read from PRINTER:)
4	Unknown hardware error (not used on Apple)
5	Lost device -- no longer on line
6	Lost file -- file is no longer in directory
7	Bad title -- illegal filename
8	No room -- insufficient space on diskette
9	No device -- volume is not on line
1Ø	No such file on specified volume
11	Duplicate file title
12	Attempt to open an already open file
13	Attempt to access a closed file
14	Bad input format -- error in reading real or integer
15	Ring buffer overflow -- input arriving too fast
16	Write-protect error -- diskette is write-protected
64	Device error -- bad address or data on diskette

In normal operation, the Compiler will generate code to perform run-time checks after each I/O operation except UNITREAD, UNITWRITE, BLOCKREAD, or BLOCKWRITE. This causes the program to get a run-time error on a bad I/O operation. Therefore if you want to check IORESULT with your own code in the program, you must disable this compiler feature by using the (*$I-*) option (see Chapter 4).

Note that IORESULT only gives a valid return the first time it is referenced after an I/O operation. If it is referenced again (without another I/O operation), it will always return Ø.

INTRODUCTION TO TEXT I/O

In addition to PUT and GET, Apple Pascal provides the standard procedures READ, READLN, WRITE, and WRITELN, collectively known as the text I/O procedures. They perform the same tasks as in standard Pascal and have the same syntax (with the addition of STRING variables). However, the details of their operation are specific to Apple Pascal and can be complicated. Also, the use of STRING variables and the distinction between TEXT and INTERACTIVE files have important effects.

The text I/O procedures can only be used with files of type TEXT or INTERACTIVE. As already mentioned, RESET makes a distinction between these two file types: when a TEXT file is RESET, a GET is automatically performed but when an INTERACTIVE file is RESET, no GET is performed. This requires READ and READLN to be rather complex procedures. Like many other complex creatures, they will behave simply if you use them simply. Therefore, this manual is written with some assumptions in mind about how they will be used. These assumptions can be translated into the following specific suggestions:

- When using the text I/O procedures don't use GET or PUT, and don't refer explicitly to the file buffer variable F^. The reason is that the text I/O procedures themselves use GET and PUT in complicated ways.

- Don't mix reading and writing operations on the same diskette textfile. If you read from a textfile, CLOSE it and reopen it before writing to it; and vice versa.

- To open an existing diskette textfile for reading, always use RESET. To open an existing diskette textfile for writing, always use REWRITE.

- Don't use READ with a STRING variable. Use READLN.

- Don't use the EOLN function with READLN, and don't use it with STRING variables.

If you follow these suggestions, the text I/O procedures will work exactly as described in the following pages. These are not rules of Pascal; there is nothing in the system that will enforce them. However, the exact details of what happens if you ignore the suggestions are beyond the scope of this chapter.

There may be situations in which these assumptions and suggestions are too restrictive. If so, you will need the complete details on how READ and READLN behave in all possible situations, as given in Appendix C.

In particular, you need the information in Appendix C if you want to mix reading and writing operations or overwrite part of an existing text file without destroying all of the original contents.

THE READ PROCEDURE

This procedure may be used only on TEXT (FILE OF CHAR) or INTERACTIVE files. It allows characters and numeric values to be read from a file without the need for explicit use of GET or explicit reference to the window variable. The form is

```
PROCEDURE READ ( [ FILEID, ] VBLs )
```

where FILEID is the identifier of a TEXT or INTERACTIVE file which must

be open. If the FILEID is omitted, INPUT is assumed. VBLs means one or more variables separated by commas. The variables may be of type CHAR, STRING, INTEGER, LONG INTEGER, or REAL. (But you should use READLN for STRING variables).

READ reads values from the file and assigns them to the variables in sequence.

READ With a CHAR Variable

For a CHAR variable, READ reads one character from the file and assigns that character to the variable. There are two special cases: Whenever the end-of-line character (ASCII 13) is READ, the value assigned to the CHAR variable is a space (ASCII 32), not a CR. Whenever EOF becomes true, the value assigned to the CHAR variable is not defined.

After the READ, the next READ or READLN will always start with the character immediately following the one just READ.

The workings of EOLN and EOF depend on whether the file is of type TEXT or INTERACTIVE. For a TEXT file, EOF is true when the last text character in the file has been READ. EOLN is true when the last text character on a line has been READ and whenever EOF is true. (A "text character" here means a character that is not the end-of-line character or the end-of-file character.)

For an INTERACTIVE file, EOF is not true until the end-of-file character has been READ. EOLN is not true until the end-of-line character at the end of the line has been READ or until EOF is true.

If you are using READ with a CHAR variable and you need to use EOLN, you may be able to simplify the situation by using READLN with a STRING variable instead; this gives you line-oriented reading without the need to check EOLN (see below).

READ With a Numeric Variable

For a variable of one of the numeric types, READ expects to read a string of characters which can be interpreted as a numeric value of the same type. Any space or end-of-line characters preceding the numeric string are skipped; and a space, end-of-line, or end-of-file is expected after the numeric string. If a numeric string is not found after skipping spaces and end-of-lines, an I/O error occurs. Otherwise, the string is converted to a numeric value and the value is assigned to the variable.

After the READ, the next READ or READLN will always start with the character immediately following the last character of the numeric string.

If the last character of the numeric string is the last character on the line, then EOLN will be true. If the last character of the numeric string is the last character in the file, then EOF and EOLN will both be true.

If nothing but spaces are found before the EOF, a value of 0 is READ.

Note that the behavior of READ with a numeric variable is exactly the same regardless of whether the file is TEXT or INTERACTIVE.

THE READLN PROCEDURE

This procedure may be used only on TEXT (FILE OF CHAR) or INTERACTIVE files. It allows line-oriented reading of characters, strings, and numeric values. The form is

```
PROCEDURE READLN ( [ FILEID, ] VBLs )
```

where FILEID is the identifier of a TEXT or INTERACTIVE file which must be open. If the FILEID is omitted, INPUT is assumed. VBLs means one or more variables separated by commas. The variables may be of type CHAR, STRING, INTEGER, LONG INTEGER, or REAL.

READLN works exactly like READ, except that after a value has been read for the last variable, the remainder of the line is skipped (including the end-of-line). After any READLN, the next READ or READLN will always start with the first character of the next line, if there is a next line. If there is no next line, EOF will be true.

READLN with a STRING variable reads all the characters up to but not including the end-of-line character. Thus repeated READLN's with a STRING variable have the effect of reading successive lines of the file as strings.

One of the most common uses of READLN with a STRING variable is to read a string of characters from the CONSOLE: device. In the following example, which is a modification of the previous example under GET and PUT, READLN is used to read a filename typed by the user:

```
PROGRAM MAKEFILE;

VAR F: FILE OF REAL;
    I: INTEGER;
    TITLE: STRING;

BEGIN
(*Ask user for title.*)
   WRITE('Type name of file: ');
(*Accept line typed by user.*)
   READLN(TITLE);
(*If title has no suffix, add .DATA suffix.*)
   IF POS('.', TITLE)=Ø THEN TITLE:=CONCAT(TITLE, '.DATA');
(*Open with REWRITE since this is a new file*)
   REWRITE(F, TITLE);
   ...
(*Remainder of program is identical to previous example.*)
```

Another useful example is given below under WRITE and WRITELN.

THE WRITE AND WRITELN PROCEDURES

These procedures may be used only on TEXT (FILE OF CHAR) or INTERACTIVE files. They allow characters, strings, and numeric values to be written to a file without the need for explicit use of PUT or explicit reference to the window variable. Also, WRITELN allows line-oriented output. The forms are

```
WRITE ( [ FILEID, ] [ ITEMs ] )
WRITELN [ ( [ FILEID, ] [ ITEMs ] ) ]
```

where FILEID is the identifier of a TEXT or INTERACTIVE file which must be open. If the FILEID is omitted, OUTPUT is assumed.

ITEMs means one or more ITEMs separated by commas. Each ITEM has one of the following forms:

```
EXPR

    or

EXPR : FIELDWIDTH

    or

EXPR : FIELDWIDTH : FRACTIONLENGTH
```

where EXPR is an expression whose value is to be written, FIELDWIDTH is an INTEGER expression which specifies the minimum number of characters to be written, and FRACTIONLENGTH is an INTEGER expression which specifies the number of digits to be written after the decimal point if EXPR is of type REAL. The default FRACTIONLENGTH is 5; the default FIELDWIDTH is 1. For a non-negative REAL value, one space is always written before the first digit; for a negative REAL value, the minus sign occupies this position.

WRITE evaluates the expressions and writes their values to the file in sequence. If EXPR is of type CHAR, STRING, or PACKED ARRAY of CHAR, WRITE writes the character(s) to the file and advances the file pointer. If a FIELDWIDTH has been given and the number of characters written is less than specified, leading spaces are added to fill the field.

If EXPR is of a numeric type, WRITE converts the value to a string of characters in standard Pascal numeric format, writes this string to the file, and advances the pointer. If the value is REAL and a FRACTIONLENGTH has been given, the specified number of digits are written after the decimal point; if no FRACTIONLENGTH is given, five decimal places are written. If necessary, the value is rounded (not truncated) to the number of decimal places available. If a FIELDWIDTH has been given and the number of characters written is less than specified, leading spaces are added to fill the field.

WRITELN works exactly like WRITE, except that after the last value has been written a return character is written to end the line. This allows line-oriented output with string expressions.

OUTPUT is the identifier of a predeclared INTERACTIVE file which can be used with WRITE and WRITELN. All characters written to OUTPUT are displayed on the console screen. When a program is writing to OUTPUT, the user may type ctrl-S to stop the output. The program halts until another character is typed, then resumes the output where it left off. Also, the user may type ctrl-F. This halts the displaying of characters on the console screen, but the program continues to run.

The following example program illustrates a number of useful techniques. It uses line-oriented I/O with STRING variables, but performs character manipulations on the STRING variables. It also shows a useful trick for opening a file for output which may or may not exist already. The effect of the program is to read the input file line by line, remove any leading periods from the lines, and write the lines out to the output file.

```
PROGRAM FLUSHPERIODS;

CONST PERIOD='.';

VAR INFILE, OUTFILE: TEXT;
    INNAME, OUTNAME, LINEBUF: STRING;
```

```
    BEGIN

(*First get the files open.*)
(*Get input filename.*)
      WRITE('Name of input file: ');
      READLN(INNAME);
(*Supply the default suffix .TEXT if needed.*)
      IF POS('.', INNAME)=Ø THEN INNAME:=CONCAT(INNAME,'.TEXT');

(*Turn off automatic error checking so program can do it.*)
      (*$I-*)
(*Input file should already exist, so open with reset.*)
      RESET(INFILE,INNAME);
(*If it doesn't work, complain and stop program.*)
      IF IORESULT<>Ø THEN BEGIN
        WRITELN('File not found.');
        EXIT(PROGRAM)
      END;
(*Turn automatic error checking back on.*)
      (*$I+*)

(*Get output filename.*)
      WRITE('Name of output file: ');
      READLN(OUTNAME);
(*Supply default suffix .TEXT if needed.*)
      IF POS('.', OUTNAME)=Ø THEN OUTNAME:=CONCAT(OUTNAME, '.TEXT');
(*Open file with rewrite.*)
      REWRITE(OUTFILE,OUTNAME);

(*Now do the job.*)
      WHILE (NOT EOF(INFILE)) AND (NOT EOF(OUTFILE)) DO BEGIN
        READLN(INFILE,LINEBUF);
        IF LENGTH(LINEBUF) >Ø THEN
          IF POS(PERIOD, LINEBUF)=1 THEN DELETE(LINEBUF, 1, 1);
        WRITELN(OUTFILE,LINEBUF)
      END;

(*Now clean up.*)
(*If the output file isn't complete...*)
      IF EOF(OUTFILE) THEN BEGIN
        WRITELN('Not enough room in output file!');
(*...Then throw it away.*)
        CLOSE(OUTFILE,PURGE)
      END
(*If it's okay, then lock it into the directory.*)
      ELSE CLOSE(OUTFILE,LOCK);
      CLOSE(INFILE)
    END.
```


THE PAGE PROCEDURE

This procedure sends a top-of-form character (ASCII 12) to the file. The form is

```
PAGE ( FILEID )
```

where FILEID is the identifier of an open file of type TEXT or INTERACTIVE.

THE SEEK PROCEDURE

This procedure allows the program to move a file pointer to any specified record in a file that is not a textfile. This allows random access to file records. The form is

```
SEEK ( FILEID , RECNUM )
```

where FILEID is the identifier of an open file that is not a textfile(i.e. not created with the .TEXT suffix), and RECNUM is an integer value interpreted as a record number in the file.

This procedure changes the file pointers so that the next GET or PUT from/to the file uses the record of FILEID specified by RECNUM. Records in files are numbered from Ø. A GET or PUT must be executed between SEEK calls since two SEEKs in a row may cause unexpected, unpredictable junk to be held in the window and associated buffers. Immediately after a SEEK, EOF will return false; a following GET or PUT will cause EOF to return the appropriate value.

The following sample program demonstrates the use of SEEK to randomly access and update records in a file:

```
    PROGRAM RANDOMACCESS;
(*Allows update of any selected record in a file.*)
    VAR
      RECNUMBER: INTEGER;
      FNAME: STRING;
      VITALS: FILE OF RECORD
                        NAME: STRING[2Ø];
                        DAY,MONTH,YEAR: INTEGER;
                        ADDRESS: STRING[5Ø];
                        ALIVE: BOOLEAN
                      END;
```

```
    BEGIN
(*Obtain filename.*)
      WRITE('Enter filename: ');
      READLN(FNAME);
(*Use RESET to preserve existing contents of file; but if it doesn't
exist, use REWRITE to create it.*)
      (*$I-*)
      RESET(VITALS, FNAME);
      IF IORESULT<>Ø THEN REWRITE(VITALS, FNAME);
      (*$I+*)

(*Repeat the following "forever," i.e. until EXIT is caused by user
typing ctrl-C and causing EOF(INPUT), or by lack of diskette space for
new records.*)
      WHILE TRUE DO BEGIN
(*Obtain record number; quit if user types ctrl-C, causing EOF.*)
        WRITE('Enter record number: ');
        READLN(RECNUMBER);
        IF EOF THEN BEGIN
                      CLOSE(VITALS, LOCK);
                      EXIT(PROGRAM)
                    END;

(*GET the specified record*)
        SEEK(VITALS,RECNUMBER);
        GET(VITALS);

(*Update the record*)
        WITH VITALS^ DO BEGIN
          WRITELN(NAME);
          WRITE('Enter correct name: ');
          READLN(NAME);
          WRITELN(DAY);
          WRITE('Enter correct day: ');
          READLN(DAY);
(*...and so forth with other fields of record.*)
        END;

(*Now SEEK the same record again, since the GET advanced the file
pointer to the next record after it got the current record into
VITALS^ *)
        SEEK(VITALS,RECNUMBER);

(*PUT updated record into file; exit if this causes EOF.*)
        PUT(VITALS);
        IF EOF(VITALS) THEN BEGIN
                              WRITELN('Not enough file space!');
                              EXIT(PROGRAM)
                            END
      END
    END.
```


THE UNITREAD AND UNITWRITE PROCEDURES

THESE ARE DANGEROUS PROCEDURES

These are the low-level procedures which do device-oriented I/O. The forms are

```
UNITREAD ( UNITNUMBER, ARRAY, LENGTH [, [BLOCKNUMBER] [, MODE]] )
UNITWRITE ( UNITNUMBER, ARRAY, LENGTH [, [BLOCKNUMBER] [, [MODE]] )
```

where:

UNITNUMBER, an integer, is the volume number of an I/O device. The Apple Pascal Operating System Reference Manual describes these numbers.

ARRAY is the name of a packed array, which may be subscripted to indicate a starting position. This is used as the starting address to do the transfers from/to. A string may also be used, but it should have a subscript greater than Ø, since the Øth element of a string contains data which usually should not be transmitted.

LENGTH is an integer value designating the number of bytes to transfer.

BLOCKNUMBER, an integer, is meaningful only when using a disk drive and is the absolute block number at which the transfer will start. If the BLOCKNUMBER is left out, Ø is assumed.

MODE, an integer in the range Ø..15, is optional; the default is Ø. It controls two UNITWRITE options which are described below. MODE has no effect on UNITREAD.

The UNITWRITE options controlled by the MODE parameter apply only to text-oriented I/O devices such as the console or a printer; they do not apply to diskette drives. Both options are enabled by default, if no MODE parameter is supplied.

One option is conversion of DLE codes. In a Pascal textfile, any leading spaces at the beginning of a line are represented by a DLE character (ASCII 16) followed by a code value which is 32 plus the number of spaces. On output to a non-block-structured device such as a printer, the DLE conversion option detects the DLE code and converts it into a sequence of spaces.

Conversion of DLE codes is disabled by a MODE value that has a one in Bit 3 (see below).

The other option is automatic line feeds. In a Pascal textfile, the end of each line is marked by the end-of-line character CR (ASCII 13) without any line-feed character. On output to a non-block-structured

device such as a printer, the automatic line-feed option inserts an LF character (ASCII 1Ø) after every CR character (ASCII 13).

Automatic line feeds are disabled by a MODE value that has a one in Bit 2 (see below).

Only Bit 2 and Bit 3 of the MODE value have any significance. Bit 2, by itself, corresponds to a value of 4, and Bit 3 by itself corresponds to a value of 8. The following values can be used to control the options:

- MODE=Ø (the default value) causes both options to be enabled.
- MODE=4 causes automatic line feeds to be disabled, while leaving DLE conversion enabled.
- MODE=8 causes DLE conversion to be disabled, while leaving automatic line feeds enabled.
- MODE=12 disables both DLE conversion and automatic line feeds.

THE UNITBUSY FUNCTION

This is a UCSD Pascal procedure used to indicate whether a specified device is busy. But since the I/O drivers on the Apple are not interrupt driven, UNITBUSY will always return the value FALSE. To test whether a character is available from the Apple keyboard, use the KEYPRESS function (see Chapter 7).

THE UNITWAIT PROCEDURE

This is a UCSD Pascal procedure which waits for a specified device to complete the I/O in progress. But since the I/O drivers on the Apple are not interrupt driven, UNITWAIT does nothing.

THE UNITCLEAR PROCEDURE

This procedure cancels all I/O operations to the specified unit and resets the hardware to its power-up state. The form is

```
UNITCLEAR ( UNITNUMBER )
```

IORESULT is set to a non-zero value if the specified unit is not present (you can use this to test whether or not a given device is present in the system). The form

```
UNITCLEAR (1)
```

flushes the type-ahead buffer for CONSOLE: and resets horizontal scrolling to full left (displays leftmost 4Ø characters on Apple's screen).

THE BLOCKREAD AND BLOCKWRITE FUNCTIONS

These functions transfer data to or from an untyped file. They return an integer value which is the number of blocks of data actually transferred. The forms are

```
BLOCKREAD  ( FILEID, ARRAYNAME, BLOCKS [, RELBLOCK] )
BLOCKWRITE ( FILEID, ARRAYNAME, BLOCKS [, RELBLOCK] )
```

where

FILEID must be the identifier of a previously declared untyped file.

ARRAYNAME is the identifier of a previously declared array. The length of the array should be an integer multiple of 512. ARRAYNAME may be indexed to indicate a starting position in the array.

BLOCKS is the number of blocks to be transferred.

RELBLOCK is the block number relative to the start of the file, the zero-th block being the first block in the file. If no RELBLOCK is specified, the reads/writes will be done sequentially. Specifying RELBLOCK moves the file pointer.

WARNING: Caution should be exercised when using these functions, as the array bounds are not heeded. EOF(FILEID) becomes true when the last block in a file is read.

The following program illustrates the use of BLOCKREAD and BLOCKWRITE.

```
PROGRAM FILEDEMO;

VAR
  BLOCKNUMBER,BLOCKSTRANSFERRED:INTEGER;
  BADIO: BOOLEAN;
  G,F: FILE;
  BUFFER:  PACKED ARRAY[Ø..511] OF CHAR;

 (* This program reads a diskfile called 'SOURCE.DATA' and copies
 the file into another diskfile called 'DESTINATION' using untyped
 files and the built-ins BLOCKREAD and BLOCKWRITE *)

BEGIN
  BADIO:=FALSE;
  RESET(G,'SOURCE.DATA');
  REWRITE(F,'DESTINATION');
  BLOCKNUMBER:=Ø;
  BLOCKSTRANSFERRED:=BLOCKREAD(G,BUFFER,1,BLOCKNUMBER);
  WHILE (NOT EOF(G)) AND (IORESULT=Ø) AND (NOT BADIO) AND
        (BLOCKSTRANSFERRED=1) DO
    BEGIN
      BLOCKSTRANSFERRED:=BLOCKWRITE(F,BUFFER,1,BLOCKNUMBER);
      BADIO:=((BLOCKSTRANSFERRED<1) OR (IORESULT<>Ø));
      BLOCKNUMBER:=BLOCKNUMBER+1;
      BLOCKSTRANSFERRED:=BLOCKREAD(G,BUFFER,1,BLOCKNUMBER)
    END;
  CLOSE(F,LOCK)
END.
```


MISCELLANEOUS BUILT-INS

THE ATAN FUNCTION

The ATAN function is simply a different identifier for the ARCTAN function of Standard Pascal. Along with the other transcendental functions, it is part of the TRANSCEND UNIT supplied with Apple Pascal (see Chapter 7).

THE LOG FUNCTION

This function returns a real value which is the logarithm (base 1Ø) of its argument. Along with the other transcendental functions, it is part of the TRANSCEND UNIT supplied with Apple Pascal (see Chapter 7). The form is

```
LOG ( NUMBER )
```

where NUMBER can be either a real or an integer value.

THE TRUNC FUNCTION

The function TRUNC will accept a LONG INTEGER as well as a REAL as an argument. Overflow will result if the absolute value of the argument exceeds MAXINT. With a REAL argument, TRUNC returns an INTEGER value formed by dropping the fractional part of the REAL value. With a LONG INTEGER value, TRUNC returns a numerically equivalent INTEGER value.

THE PWROFTEN FUNCTION

This function returns a real value which is 1Ø to a specified (integer) power. The form is

```
PWROFTEN ( EXPONENT )
```

where EXPONENT is an integer value in the range Ø..37. This function returns the value of 1Ø to the EXPONENT power.

THE MARK AND RELEASE PROCEDURES

The Standard Pascal procedure DISPOSE is not provided in Apple Pascal. Instead, the MARK and RELEASE procedures are used for returning dynamic memory allocations to the system. The forms are

```
MARK ( HEAPPTR )
RELEASE ( HEAPPTR )
```

where HEAPPTR is of type ^INTEGER and is called by reference in the MARK procedure. MARK sets HEAPPTR to the value of the system's current top-of-heap pointer. RELEASE sets the system's top-of-heap pointer to the value of HEAPPTR.

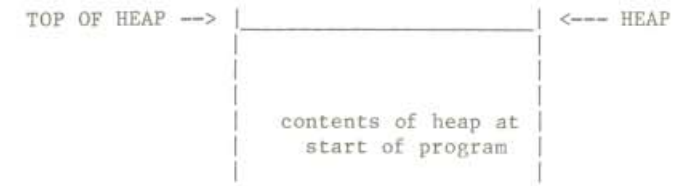
The process of recovering memory space described below is only an approximation to the function of DISPOSE as one cannot explicitly ask that the storage occupied by one particular variable be released by the system for other uses.

Variables created by the standard procedure NEW are stored in a stack-like structure called the "heap". The following program is a simple demonstration of how MARK and RELEASE can be used to change the size of the heap.

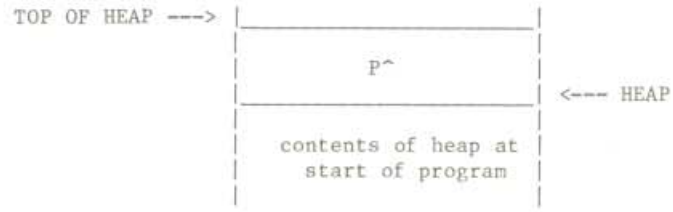
```
PROGRAM SMALLHEAP;

TYPE  PERSON=
        RECORD
          NAME: PACKED ARRAY[Ø..15] OF CHAR;
          ID: INTEGER
        END;

VAR P: ^PERSON;
    HEAP: ^INTEGER;

BEGIN
  MARK(HEAP);
  NEW(P);
  P^.NAME:='FARKLE, HENRY J.';
  P^.ID:= 999;
  RELEASE(HEAP)
END.
```

The program shows a particularly handy method for deliberately accessing the contents of memory which is otherwise inaccessable. It first calls MARK to place the address of the current top of heap into the variable HEAP.

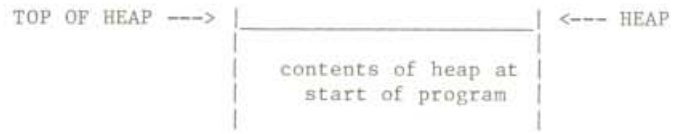
Below is a pictorial description of the heap at this point in the program's execution:



Next the program calls the standard procedure NEW and this results in a new variable P^ which is located in the heap as shown in the diagram below:



After the RELEASE the heap is as follows:



Once the program no longer needs the variable P^ and wishes to "release" this memory space to the system for other uses, it calls RELEASE which resets the top of heap to the address contained in the variable HEAP.

If NEW had been called several times between the calls to MARK and RELEASE, the storage occupied by several variables would have been RELEASEd at once. Note that because of the stack nature of the heap it is not possible to release the memory space used by a single item in the middle of the heap.

Careless use of MARK and RELEASE can leave "dangling pointers", pointing to areas of memory which are no longer part of the defined heap space.

THE HALT PROCEDURE

This procedure generates a HALT opcode that, when executed, causes a non-fatal run-time error to occur. The form is

```
HALT
```

For a more orderly way to terminate program execution, see EXIT below.

THE EXIT PROCEDURE

The EXIT procedure causes an orderly exit from a procedure or function, or from the program itself. The forms are

```
EXIT(procedurename)
EXIT(programname)
EXIT(PROGRAM)
```

In the first form shown, EXIT accepts as its single parameter the identifier of a procedure or function to be exited. Note that this need not be the procedure or function in which the EXIT statement occurs. EXIT follows the trail of procedure or function calls back to the procedure or function specified; each procedure or function in the trail is exited. If the specified procedure is recursive, the most recent invocation of that procedure will be exited.

When a procedure or function is exited via EXIT, any files local to it are automatically closed, just as if it had terminated normally.

The use of EXIT to exit a function can cause the function to return an undefined value if no assignment to the function identifier is made before EXIT is executed.

When the program name or the reserved word PROGRAM is used as the parameter for EXIT, EXIT brings the program to an orderly halt.

THE MEMAVAIL FUNCTION

This function returns the number of words currently between the top-of-stack and top-of-heap. This can be interpreted as the amount of memory available at that time. The form is

```
MEMAVAIL
```


THE GOTOXY PROCEDURE

This procedure sends the cursor to a specified position on the screen. The form is

```
GOTOXY ( XCOORD , YCOORD )
```

where XCOORD and YCOORD are integer values interpreted as X (horizontal) and Y (vertical) coordinates. XCOORD must be in the range Ø through 79; YCOORD must be in the range Ø through 23. The cursor is sent to these coordinates. The upper left corner of the screen is assumed to be (Ø,Ø).

This procedure is written to work with the Apple II's screen. If you wish to use an external terminal, you will need to bind in a new GOTOXY using the BINDER package described in the Pascal Operating System Manual.

THE TREESEARCH FUNCTION

This a fast function for searching a binary tree that has a particular kind of structure. The form is

```
TREESEARCH(ROOTPTR, NODEPTR, NAME)
```

where ROOTPTR is a pointer to the root node of the tree to be searched, NODEPTR is a pointer variable to be updated by TREESEARCH, and NAME is the identifier of a PACKED ARRAY[1..8] OF CHAR which contains the 8-character name to be searched for in the tree.

The nodes of the binary tree are assumed to be linked records of the form

```
NODE=RECORD
       NAME: PACKED ARRAY[1..8] OF CHAR;
       LEFTLINK, RIGHTLINK: ^NODE;

           ...(*other fields can be anything*)...

     END;
```

The type name and the field names are not important; TREESEARCH only assumes that the first eight bytes of the record contain an 8-character name and are followed by two pointers to other nodes.

It is also assumed that names are not duplicated within the tree and are assigned to nodes according to an alphabetical rule: for a given node, the name of the left subnode is alphabetically less than the name of the node, and the name of the right subnode is alphabetically greater than the name of the node. Finally, any links that do not point to other nodes should be NIL.

TREESEARCH can return any of three values:

Ø: The NAME passed to TREESEARCH has been found in the tree. NODEPTR now points to the node with the specified name.

1: The NAME is not in the tree. If it is added to the tree, it should be the right subnode of the node pointed to by NODEPTR.

-1: The NAME is not in the tree. If it is added to the tree, it should be the left subnode of the node pointed to by NODEPTR.

The TREESEARCH function does not perform any type checking on the parameters passed to it.

BYTE-ORIENTED BUILT-INS

These procedures and functions are all byte-oriented. The system does not protect itself from them, as no range checking of any sort is performed on the parameters and no type checking is performed on the source and destination parameters. Read the descriptions carefully before trying them out. Also, some machine dependencies may lurk in the implementations.

THE SIZEOF FUNCTION

This function returns an integer value, which is the number of bytes occupied by a specified variable, or by any variable of a specified type. SIZEOF is particularly useful for FILLCHAR, MOVERIGHT, and MOVELEFT built-ins (see below). The form is

```
SIZEOF ( IDENTIFIER )
```

where IDENTIFIER is either a type identifier or a variable identifier.

THE SCAN FUNCTION

This function scans a range of memory bytes, looking for a one-character target. The target can be a specified character, or it can be any character that does not match the specified character. SCAN returns an integer value, which is the number of bytes scanned. The form is

```
SCAN ( LIMIT , PEXPR , SOURCE )
```

where

LIMIT is an integer value which gives the maximum number of bytes to scan. If LIMIT is negative, SCAN will scan backward. If SCAN fails to find the specified target, it will return the value of LIMIT.

PEXPR is a "partial expression" which specifies the target of the scan. PEXPR takes one of the following forms:

```
= CH          (target is a character equal to CH)
<> CH         (target is a character not equal to CH)
```

where CH stands for any expression that yields a result of type char.

SOURCE is a variable of any type except a file type. The first byte of the variable is the starting point of the scan.

SCAN terminates when it finds the target or when it has scanned LIMIT bytes. It then returns the number of bytes scanned. If the target is found at the starting point, the value returned will be zero. If LIMIT is negative, the scan will go backward and the value returned will also be negative.

Examples: Suppose that DEM is declared as follows:

```
VAR DEM: PACKED ARRAY [0..100] OF CHAR;
```

and then the first 53 elements of DEM are loaded with the characters

```
.....THE PTERO IS A MEMBER OF THE PTERODACTYL FAMILY.
```

We then have the following:

```
SCAN(-26,=':',DEM[30])      will return -26

SCAN(100,<>'.',DEM)         will return 5

SCAN(15,=' ',DEM[5])        will return 3.
```

THE MOVELEFT AND MOVERIGHT PROCEDURES

These procedures do mass moves of a specified number of bytes. The forms are

```
MOVELEFT ( SOURCE , DESTINATION , COUNT )
MOVERIGHT ( SOURCE , DESTINATION , COUNT )
```

where SOURCE and DESTINATION are two variables of any type except a file type. The first byte of SOURCE is the beginning of the range of bytes whose values are copied. The first byte of DESTINATION is the beginning of the range of bytes that the values are copied into. COUNT is an integer and specifies the number of bytes moved.

MOVELEFT starts from the left end of the SOURCE range. It proceeds from left to right, copying bytes into DESTINATION, starting at the left end of the DESTINATION range.

MOVERIGHT starts from the right end of the SOURCE range. It proceeds from right to left, copying bytes into DESTINATION, starting at the right end of the DESTINATION range.

The reason for having both of these is that the SOURCE and DESTINATION ranges may overlap. If they overlap, the order in which bytes are moved is critical: each byte must be moved before it gets overwritten by another byte.

In general this consideration applies when SOURCE and DESTINATION are subarrays of the same PACKED ARRAY OF CHAR. If bytes are being moved to the right (DESTINATION has a higher subscript than SOURCE), use MOVERIGHT. If bytes are being moved to the left (DESTINATION has a lower subscript than SOURCE), use MOVELEFT.

THE FILLCHAR PROCEDURE

This procedure fills a specified range of memory bytes with a specified character value. The form is

```
FILLCHAR ( DESTINATION , COUNT , CHARACTER )
```

where DESTINATION is a variable of any type except a file type. The first byte of DESTINATION is the beginning of the range of bytes to be filled. COUNT is an integer value and specifies the number of bytes to be filled. CHARACTER is a character value to be copied into each byte in the specified range.

SUMMARY

STRING BUILT-INS

Integer-Valued Functions:

LENGTH (STRG) returns length of string.
POS (SUBSTRG , STRG) returns index of first occurence of SUBSTRG within STRG.

String-Valued Functions:

CONCAT (STRGs) returns concatenation of strings.
COPY (STRG , INDEX , COUNT) returns a substring of STRG.

Procedures:

DELETE (STRG , INDEX , COUNT) deletes a substring of STRG.
INSERT (SUBSTRG , STRG , INDEX) inserts a substring into STRG.
STR (LONG , STRG) converts integer or long integer to string of decimal digits.

INPUT AND OUTPUT BUILT-INS

Opening and Closing Files:

RESET (FILEID [, TITLE]) opens existing diskette file, or resets pointers to beginning if already open.
REWRITE (FILEID , TITLE) opens new diskette file.
CLOSE (FILEID [, OPTION]) closes file. OPTION may be LOCK, NORMAL, PURGE, or CRUNCH. Default is NORMAL.

File Pointer Status:

EOF [(FILEID)] boolean, true if end of file has been reached or file is closed. Default FILEID is INPUT.
EOLN [(FILEID)] boolean, true if end of line has been reached. Default FILEID is INPUT.
SEEK (FILEID , INTEGER) moves file pointer to specified record.

Typed File I/O:

GET (FILEID) reads current file record into window & advances file pointer. Default FILEID is INPUT.
PUT (FILEID) writes window into current file record & advances file pointer. Default FILEID is OUTPUT.
IORESULT returns an integer value which depends on status of most recent I/O operation. Value is zero for OK completion.
READ ([FILEID,] VBLs) where VBLs means one or more variables separated by commas. Successive values are read from file into variables. Default FILEID is INPUT. FILEID must be of type TEXT (FILE OF CHAR) or INTERACTIVE.
READLN ([FILEID,] VBLs) Like READ, but skips to beginning of next line after reading value for last VBL.
WRITE ([FILEID,] [EXPRs]) where EXPRs means one or more expressions separated by commas. Each EXPR may also be followed by field width and number of decimal places. Expression values are written to successive file records. Default FILEID is OUTPUT. FILEID must be of type TEXT (FILE OF CHAR) or INTERACTIVE.
WRITELN [([FILEID,] [EXPRs])] Like WRITE, but writes an end-of-line after last EXPR value.
PAGE (FILEID) writes a top-of-form (ASCII 12).

Device I/O:
These built-ins are described in detail in the text.

```
UNITREAD ( UNITNUMBER , ARRAY , LENGTH [, [BLOCKNUMBER] [, MODE]] )
UNITWRITE ( UNITNUMBER , ARRAY , LENGTH [, [BLOCKNUMBER] [, MODE]] )
UNITBUSY ( UNITNUMBER ) : BOOLEAN
UNITWAIT ( UNITNUMBER )
UNITCLEAR ( UNITNUMBER )
```

Untyped File I/O:
These built-ins are described in detail in the text.

```
BLOCKREAD (FILEID, ARRAY, BLOCKS [, RELBLOCK]) : INTEGER
BLOCKWRITE (FILEID, ARRAY, BLOCKS [, RELBLOCK]): INTEGER
```

MISCELLANEOUS BUILT-INS

ATAN (NUMBER) returns a REAL value. This is the ARCTAN function of Standard Pascal. NUMBER may be REAL or INTEGER.

LOG (NUMBER) returns a REAL value, the log base 1Ø of NUMBER. NUMBER may be REAL or INTEGER.

TRUNC (NUMBER) returns an INTEGER value. This is like Standard Pascal except that NUMBER may be LONG INTEGER instead of REAL. NUMBER may not exceed MAXINT.

PWROFTEN (EXPONENT) returns a REAL value which is 1Ø to the EXPONENT power. EXPONENT is an INTEGER in the range Ø..37.

MARK (HEAPPTR) where HEAPPTR is of type ^INTEGER. HEAPPTR is called by name and is set to current top-of-heap.

RELEASE (HEAPPTR) where HEAPPTR is of type ^INTEGER. The current top-of-heap pointer is set to HEAPPTR.

HALT causes non-fatal run-time error; halts program.

EXIT causes orderly exit from procedure, function, or program.

MEMAVAIL returns an INTEGER value, the number of words between top-of-stack and top-of-heap.

GOTOXY (XCOORD , YCOORD) moves screen cursor to specified coordinates. XCOORD is an INTEGER in the range Ø..79 and YCOORD is an INTEGER in the range Ø..23.

TREESEARCH (ROOTPTR , NODEPTR , NAME) searches for NAME in a binary tree. See text for details.

BYTE-ORIENTED BUILT-INS

These built-ins are described in detail in the text.

SIZEOF (VARIABLE OR TYPE IDENTIFIER)
SCAN (LIMIT , PEXPR , SOURCE)
MOVELEFT (SOURCE , DESTINATION , COUNT)
MOVERIGHT (SOURCE , DESTINATION , COUNT)
FILLCHAR (DESTINATION , COUNT , CHARACTER)

CHAPTER 4
THE PASCAL COMPILER

INTRODUCTION

The purpose of the Apple Pascal Compiler is to translate the text of a Pascal program into the compressed P-code version of the program. This P-code is the "machine language" of the UCSD Pascal interpreter or "pseudo-machine," described in the Apple Pascal Operating System Manual.

Complete details on operation of the Compiler are in the Pascal Operating System Reference Manual; the following two sections on diskette files needed and on using the Compiler are somewhat abridged.

DISKETTE FILES NEEDED

To operate the Pascal Compiler, you need the following diskette files:

Textfile to be Compiled	(Any diskette, any drive; default is boot diskette's text workfile SYSTEM.WRK.TEXT, any drive)
SYSTEM.COMPILER	(Any diskette, any drive)
SYSTEM.LIBRARY	(Boot diskette, any drive; required only if any of the UNITs in the system library are USEd by the program. See Chapter 5.)
Other Libraries	(Any diskette, any drive; required if any UNITs not in the system library are USEd by the program being compiled. See Chapter 5.)
SYSTEM.EDITOR	(Any diskette, any drive; optional; to fix errors found by Compiler)
SYSTEM.SYNTAX	(Boot diskette, any drive; optional messages given on entering Editor)

In addition to the above files, the following files may be needed if you are invoking the Compiler automatically via the R(un command (see Apple Pascal Operating System Reference Manual for details):

SYSTEM.LINKER
SYSTEM.PASCAL
SYSTEM.CHARSET

One-drive note: The files SYSTEM.COMPILER, SYSTEM.EDITOR, and SYSTEM.SYNTAX are all on diskette APPLEØ:, which is the normal one-drive boot diskette. If you have been working on a program in the Editor, and U(pdating the workfile, your boot diskette has all the files needed to R(un or C(ompile the workfile. If you wish to R(un or C(ompile a textfile that is not already on the boot diskette, use the Filer's T(ransfer command to transfer that textfile onto your boot diskette before compiling. If your program requires Linking to external routines, see the Apple Pascal Operating System Manual.

Multi-drive note: The files SYSTEM.EDITOR and SYSTEM.SYNTAX are both on diskette APPLE1:, which is the normal multi-drive boot diskette. The file SYSTEM.COMPILER is on diskette APPLE2:, which is normally kept in drive volume #5: in a multi-drive system. With APPLE1: in the boot drive and APPLE2: in a non-boot drive, your system has all the files needed to R(un or C(ompile the workfile.

Two-drive note: If you wish to R(un or C(ompile a textfile that is not already on APPLE1: or APPLE2:, and your system has only two drives, use the Filer's T(ransfer command to transfer that textfile onto either APPLE1: or APPLE2: before compiling. Another possibility for two-drive systems is to make APPLEØ: your boot diskette (just put APPLEØ: in the boot drive and press the Apple's RESET key). This frees your second drive to hold a source or destination diskette for compilations, saving you from T(ransferring the source file onto APPLE1: or APPLE2:. APPLEØ: does not contain SYSTEM.LINKER; if your program requires Linking to external routines, use APPLE1: and APPLE2:.

USING THE COMPILER

The Compiler is invoked by typing C for C(ompile or R for R(un from the outermost Command level of the Pascal system. The screen immediately shows the message

```
COMPILING...
```

The Compiler automatically compiles the .TEXT part of the workfile and saves the resulting code (if compilation is successful) as the .CODE part of the workfile. If there is a workfile, but you do not wish to compile that file, use the Filer's N(ew command to clear away the workfile before compiling. If no workfile is available, you are prompted for a source filename:

```
COMPILE WHAT TEXT?
```

You should respond by typing the name of the text file that you wish to have compiled. Do NOT type the suffix .TEXT -- that suffix is automatically supplied by the Compiler, in addition to any suffix you may specify.

Next, if there is no workfile, you will be asked for the name of the file where you wish to save the compiled version of your program:

```
TO WHAT CODEFILE?
```

If you simply press the RETURN key the command will not be terminated, as you might expect. Instead, the source file will be compiled and the compiled version of your program will be saved on the boot diskette's

workfile SYSTEM.WRK.CODE. This is handy if you then wish to R(un the program.

If you want the compiled version of your program to have the same name as the text version of your program (of course, the suffix will be .CODE instead of .TEXT), just type a dollar sign and press the RETURN key. This is a handy feature, since you will usually want to remember only one name for both versions of your program. The dollar sign repeats your entire source file specification, including the volume identifier, so do NOT specify the volume identifier before typing the dollar sign. Note that this use is different from the use of the dollar sign in the Filer.

If you want your program stored under another filename, type the desired filename. Do NOT type the suffix .CODE -- that suffix is automatically supplied by the Compiler, in addition to any suffix you may specify.

By default, the compiler places the code file at the beginning of the largest unused space on the diskette. To override this, you can give a size specification with the filename. In this case, you DO type the suffix .CODE, followed by the number of blocks in square brackets, followed by a period:

```
TO WHAT CODEFILE? MYPROG.CODE[8].
```

The period at the end prevents the system from adding the .CODE prefix after the size specification. The size specification [8] causes the code file to be placed in the first location on the diskette where at least 8 blocks are available.

While the compiler is running, messages on the screen show the progress of the compilation as in the following example:

```
PASCAL COMPILER II.1 [B2B]
<   Ø>.....
TUNAFISH  [  2334 WORDS]
<   6>........
14 LINES
SMALLEST AVAILABLE SPACE = 2334 WORDS
```

The identifiers appearing on the screen are the identifiers of the program and its procedures. The identifier for a procedure is displayed at the moment when compilation of the procedure body is started.

The numbers within [ ] indicate the number of (16-bit) words available for symbol table storage at that point in the compilation. If this number ever falls below 55Ø, the compiler will fail. You must then put the swapping option (described below) into your program and recompile.

The numbers enclosed within < > are the current line numbers. Each dot on the screen represents one source line compiled.

If the Compiler detects an error in your program, the screen will show the text preceding the error, an error number, and a marker <<<< pointing to the symbol in the source where the error was detected. The following is an example:

```
[ <<<<
LINE 9, ERROR 18: <SP>(CONTINUE), <ESC>(TERMINATE), E(DIT
```

This shows that the bracket [is an illegal symbol at this point in the program. You have three options when you see a message like this. Pressing the spacebar instructs the Compiler to continue the compilation, in case you want to find more of the errors right now. Pressing the ESC key causes termination of the compilation and return to the Command level.

Typing E sends you to the Editor, which automatically reads in the workfile, ready for editing. If you were not compiling the workfile, the Editor requests the name of the file you were compiling. You should respond by typing the filename of the file you were compiling, and that file will then be read into the Editor. When the correct file has been read into the Editor, the top line of the screen displays the error message (or number, if SYSTEM.SYNTAX was not available) and the cursor is placed at the symbol where the error was detected.

If SYSTEM.SYNTAX is not available, you can look up the error in Table 6 of Appendix B. (You may wish to delete the file SYSTEM.SYNTAX to obtain more diskette space.)

THE COMPILER OPTIONS

COMPILER OPTION SYNTAX

Compiler options (see the following section for details) are placed in the text to be compiled, and take effect when the Compiler arrives at the option during compilation.

Compiler options look like a special kind of comment, and take the following form:

```
(*$option*)
```

The Compiler treats material between (*$ and *) as a compiler option. As shown below, there must be no spaces in (*$ or immediately after the $ character:

```
(*$G-*)          This is a compiler option.
(* $G-*)         This is a comment.
(*$ G-*)         This is a comment.
```

Several options can be combined in one set of (*$...*) brackets, by separating the options with commas (don't add extra spaces):

(*$option,option*) Example: (*$I-,S+,G-*)

You can't do this with the options that involve names or strings of characters.

A given option may be turned on or off at any point in the compilation. The compilation is affected only from the point where the option is turned on until the point where the option is turned off again. Thus you can turn an option on (or off) just during the compilation of a particular routine in your program.

Some options require a filename immediately following the option letter, instead of the usual + or -. In this case, all characters between the option letter and the closing *) are taken as the filename, except that blanks preceding or following the filename are ignored. Therefore, the filename must be the last item before the *). If the first character of a filename is + or -, you must place a blank between the option letter and the filename. For examples of specifying a filename, see the section describing the Include-file mechanism.

In Apple Pascal, curly brackets { and } are equivalent to the normal comment or option delimiters (* and *). The curly brackets cannot be generated by the Apple keyboard, so no confusion exists for programs written on the Apple computer. However, other terminals may be able to generate the curly brackets in programs. These programs will be executed correctly on the Apple, but the curly brackets will be displayed on Apple's screen as square brackets [and] .

THE "COMMENT" OPTION

This option consists of the letter C and a line of text. The text is placed, character for character, in Block Ø of the codefile (where it will not affect program operation). The purpose of this is to allow a copyright notice or other comment to be embedded in the codefile. Example:

(*$C COPYRIGHT ALLUVIAL O. FANSOME 1979*)

The Comment option must precede the heading statement of the program.

THE "GOTO STATEMENTS" OPTION

Tells the Compiler whether to allow or forbid the use of the Pascal GOTO statement within a program.

Default value: G-

(*$G+*) Allows the use of the GOTO statement.

(*$G-*) Causes the Compiler to treat a GOTO as an error.

Teachers sometimes use the G- option to keep novice programmers from using the GOTO statement where more structured approaches using FOR, WHILE, or REPEAT statements would be more appropriate.

THE "IO CHECK" OPTION

This option tells the compiler whether or not to create error-checking code after each structured file I/O statement (not the BLOCK or UNIT I/O statements).

Default value: I+

(*$I+*) Instructs the Compiler to generate code after each statement which performs any I/O, in order to check that the I/O operation was accomplished successfully. In the case of an unsuccessful I/O operation, the program will be terminated with a run-time error.

(*$I-*) Instructs the Compiler not to generate any I/O-checking code. In the case of an unsuccessful I/O operation, the program is not terminated with a run-time error.

The (*$I-*) option is useful for programs where I/O checking is not desirable, or which do their own checking via the IORESULT function. The program can then detect and report the I/O errors, without being terminated abnormally with a run-time error. However, the disadvantage of setting the (*$I-*) option is that I/O errors, (and possibly severe program bugs), may go undetected.

THE "INCLUDE FILE" OPTION

The syntax for instructing the Compiler to include another source file into the compilation is as follows:

(*$I filename *)

All characters between (*$I and *) are taken as the filename of the source file to be included. Thus, the filename must be the last item

before the *). Spaces preceding the filename and following it are ignored.

Note that if the first character of a filename is + or -, you MUST place a blank space between (*$I and the filename. Also, you may not use the * or *: notation in the filename to specify the boot diskette.

If the initial attempt to open the file which is being included (also called the "include file") fails, the Compiler concatenates the suffix .TEXT to the filename and tries again. If this second attempt fails, or if some I/O error occurs while reading the include file, the Compiler responds with a fatal error message and terminates its operation.

If the include file option occurs within the body of a procedure or within the body of the main program, the includ file must not contain any USES, LABEL, CONST, TYPE, or VAR declarations. Otherwise, the compiler accepts include files which contain such declarations even though the declarations of the original program have already been compiled.

The Compiler cannot keep track of nested include options, i.e. an include file must not contain an include file option. This results in a fatal Compiler error.

The include file option makes it easier to compile large programs without having the entire source in one very large file. This is especially useful when the combined file would be too large to edit in the existing Editor's buffer.

THE "LISTING" OPTION

Controls whether the Compiler will generate a program listing.

Default value: L-

(*$L+*) Instructs the Compiler to save a compiled listing on the boot diskette under the filename SYSTEM.LST.TEXT.

(*$L-*) Tells Compiler to make no compiled listing.

(*$L filename*) Tells Compiler to save compiled listing in the specified file.

For example, the following will cause the compiled listing to be sent to the printer:

```
(*$L PRINTER:*)
```


The following will cause the compiled listing to be sent to a diskfile called DEMO1.TEXT on the diskette named MYDISK:

```
(*$L MYDISK:DEMO1.TEXT *)
```

The specified filename, which must be the last item before the *), is used exactly as typed. No suffix is added. Note that a diskette listing file may be edited just like any other text file, provided the filename which is specified contains the suffix .TEXT.

In the compiled listing, the Compiler places next to each source line the line number, segment number, procedure number, and the number of bytes or words (bytes for code, words for data) required by that procedure's declarations or code to that point. The Compiler also indicates whether the line lies within the actual code to be executed or is a part of the declarations for that procedure by printing a "D" for declaration and an integer Ø..9 to designate the level of statement nesting within the code part.

Here is a sample listing, to which column headings have been added:

```
Source  Segment  Procedure   Lexical  Byte number          Program
 line    number    number : level    within procedure      text

   1       1         1:D        1    (*$L SCRATCH:LIST1.TEXT*)
   2       1         1:D        1
   3       1         1:D        1    PROGRAM DOCTOR;
   4       1         1:D        3    VAR DAY,CURE:INTEGER;
   5       1         1:D        5
   6       1         2:D        1    PROCEDURE DOSE;
   7       1         2:Ø        Ø    BEGIN
   8       1         2:1        Ø      WRITE('AN APPLE A DAY');
   9       1         2:1       26      WRITE(' AND ')
  1Ø       1         2:Ø       43    END;
  11       1         2:Ø       56
  12       1         3:D        1    PROCEDURE TREATMENT;
  13       1         3:Ø        Ø    BEGIN
  14       1         3:1        Ø      FOR CURE:=1 TO 4 DO
  15       1         3:2       11      BEGIN
  16       1         3:3       11        DOSE
  17       1         3:2       11      END
  18       1         3:Ø       13    END;
  19       1         3:Ø       34
  2Ø       1         1:Ø        Ø    BEGIN
  21       1         1:1        Ø      FOR DAY:=Ø TO 25 DO
  22       1         1:2       13      BEGIN
  23       1         1:3       13        TREATMENT;
  24       1         1:3       15        WRITELN('')
  25       1         1:2       35      END
  26       1         1:Ø       35    END.
```

The information given in the compiled listing can be very valuable for debugging a large program. A run-time error message will indicate the segment number, procedure number, and the offset (byte number within the

current procedure) where the error occurred.

Here is a sample run-time error message:

```
EXEC ERR # 1Ø
S# 1, P# 7, I# 56
TYPE <SPACE> TO CONTINUE
```

where S# is the segment number, P# is the procedure number, and I# is the byte number in that procedure where the error occurred.

THE "NOLOAD" OPTION

This option prevents the code of a UNIT used by the program (see Chapter 5) from being kept in memory when the program is executed. Instead, the UNIT's code is in memory only when some portion of it is active.

Default value: N-

(*$N+*) UNIT code will be loaded only when active.

(*$N-*) UNIT code will be loaded as soon as program begins executing.

The (*$N*) option should be placed at the beginning of the main program. Note that use of the (*$N+*) option does not prevent the initialization portion of a UNIT from being executed.

THE "PAGE" OPTION

If a listing is being produced, the P option causes one form-feed (ASCII 12) to be inserted into the text of the listing, just before the line containing the P option. For example, if your program contains the line

```
(*$P*)
```

that line will appear at the top of a new page when you print the program's compiled listing.

THE "QUIET COMPILE" OPTION

The Q Compiler option is the "quiet compile" option which can be used to suppress the screen messages that tell the procedure names and line numbers and detail the progress of the compilation.

Default value: Q-

(*$Q+*) Causes the Compiler to suppress output to the screen.

(*$Q-*) Causes the Compiler to send procedure name and line number messages to the screen.

This is mostly useful when the CONSOLE: device is not the Apple's TV or monitor screen, for example if you are using a printing terminal. In normal use with your Apple, this option is not needed.

THE "RANGE CHECK" OPTION

With the (*$R+*) option, the Compiler will produce code which checks on array and string subscripts and on assignments to variables of subrange and string types.

Default value: R+

(*$R+*) Turns range checking on.

(*$R-*) Turns range checking off.

Note that programs compiled with the (*$R-*) option selected will run slightly faster. However if an invalid index occurs or an invalid assignment is made, the program will not be terminated with a run-time error. Since you should never assume that a program is absolutely correct, use (*$R-*) only when speed is critical.

THE "RESIDENT" OPTION

This option forces the code of a specified UNIT or SEGMENT procedure to be kept in memory, for as long as the procedure that contains the option is active. It can thus override the automatic swapping out of a SEGMENT PROCEDURE or FUNCTION (see Chapter 5), and the automatic swapping out of a UNIT caused by the NOLOAD option (see above). For example, suppose that MOBY is a large SEGMENT PROCEDURE. Normally it is in memory only when it is active (thus allowing the memory space to be used for something else). But another procedure, RATS, calls MOBY repeatedly. We don't want the disk drive to be whizzing MOBY in and out of memory each time RATS calls it, so we make MOBY a "resident procedure" within RATS:

```
PROCEDURE RATS (HATS, BATS, CATS:INTEGER);
  VAR FOON, MOON: STRING;
  BEGIN
    (*$R MOBY*)
    .
    .
    .
```

Now MOBY will be kept in memory as long as RATS is active. The resident option must immediately follow the BEGIN that starts the procedure body.

The resident option is also useful in connection with the noload option described above.

THE "SWAPPING" OPTION

This option determines whether or not the Compiler operates in "swapping" mode. There are two main parts of the Compiler: one processes declarations; the other handles statements. In the S+ swapping mode, only one of these parts is in main memory at a time. This makes about 3900 additional words available for symbol-table storage at the cost of slower compilation speed (approximately 300 lines/minute in S- mode, versus about 150 lines/minute in S+ mode) because of the overhead of swapping the Compiler segments in from disk. This option must occur before the Compiler encounters any Pascal syntax.

Default value: S-

(*$S+*) Puts Compiler in swapping mode.

(*$S-*) Puts Compiler in non-swapping mode.

(*$S++*) Compiler does even more swapping than with the S+ option. The program compiles still more slowly, but still more room is left in memory for symbol-table storage.

The S+ option should be used when compiling a UNIT.

THE "USER PROGRAM" OPTION

This option determines whether this compilation is a user program compilation, or a compilation of a system program.

Default value: U+

(*$U+*) Informs the Compiler that this compilation is to take place on the user program lex level.

(*$U-*) Tells the Compiler to compile the program at the system lex level. This setting of the U compiler option also causes the following options to be set: R-, G+, I-

NOTE: The U- option will generate programs that do not behave as expected. Not recommended for non-systems work unless you know its method of operation.

THE "USE LIBRARY" OPTION

This option consists of the letter U and a filename. The named file becomes the library file in which subsequent USEed UNITs are sought. The specified filename, which must be the last item before the *), is used exactly as typed. No suffix is added.

The default filename for the library is SYSTEM.LIBRARY, on the boot diskette. If any USEd UNITs are in the boot diskette's SYSTEM.LIBRARY, and you refer to those UNITs first, you do not need the Use-library Compiler option for those UNITs. See Chapter 5 for more details on UNITs.

Following is an example of a valid USES clause employing the U filename Compiler option:

```
USES UNIT1,UNIT2, (*FOUND IN *SYSTEM.LIBRARY*)
(*$U MYDISK:A.CODE *) UNIT3,
(*$U APPLE1:B.LIBRARY *)
UNIT4,UNIT5;
```

Note: In a U filename option, you may not use the * or *: notation to specify the boot diskette.

Some programs require the Compiler to access another diskette file -- for example, an "include" file. When this is done, 2K of memory is required for the diskette directory. If the program is very large, this additional memory is not available and the compilation fails. If this happens to you, try the following technique:

Use the Filer command M(ake to create a 4-block file named SYSTEM.SWAPDISK on the same diskette that contains the Compiler. Now, when the Compiler reads a diskette file during compilation, it will write out 2K of information from memory to SYSTEM.SWAPDISK, thus freeing 2K of memory for the diskette directory. When the diskette directory is no longer needed, the 2K of information is read back into memory from SYSTEM.SWAPDISK.

COMPILER OPTION SUMMARY

All Compiler options are placed in the source text in "dollar-sign comments":

```
(*$option*)                Examples: (*$G-*)
                                     (*$I TURTLE.TEXT *)
```

Compiler-option specifications may be combined in one set of (*$...*) brackets:

```
(*$option,option*)         Example: (*$F-,S+,G+*)
```

If a filename is specified, it must be the last item before the *).

```
C             Following characters are placed directly into codefile.

G+            Allows GOTO statements.
G-            Forbids GOTO statements (default).

I+            Generates I/O-checking code (default).
I-            No I/O checking.
I filename    Includes named source file in compilation.

L+            Sends compiled listing to SYSTEM.LST.TEXT, on boot disk.
L-            Makes no compiled listing (default).
L filename    Sends compiled listing to named file.

N+            Prevents UNITs from being loaded until activated.
N-            Loads UNITs immediately when program runs (default).

P             Inserts a page-feed into compiled listing.

Q+            Suppresses screen messages.
Q-            Sends procedure names and line numbers to CONSOLE: (default)

R+            Generates range-checking code (default).
R-            No range checking.
R name        Keeps named procedure loaded while current one is active.

S+            Puts Compiler in swapping mode.
S++           Compiler does even more swapping.
S- Non-swapping mode.

U+            Compiles user program (default).
U-            Compiles system program.
U filename    Specifies name of library file for finding UNITs.
```


CHAPTER 5
PROGRAMS IN PIECES

INTRODUCTION

Apple Pascal supports the separation of procedures and functions, or groups of them, from the main program. When you are developing a large or complex program, this can be very useful as it allows you to reduce the size of code files, to reduce the memory space used by the program, and to use a set of procedures and functions in more than one program.

Separation can be achieved both at the P-code level and at the source-language level. At the P-code level, any procedure or function can be designated as a SEGMENT. The result is that its code is not loaded into memory until it is called by some other part of the program. As soon as the SEGMENT procedure or function is no longer active it is "swapped out;" that is, its memory space is made available for some other use such as dynamic memory allocation or swapping in another SEGMENT. This technique is sometimes called "overlaying."

At the source-language level, a group of one or more procedures or functions can be compiled separately as a UNIT. The result of compiling a UNIT is a library file; it can either be used directly or incorporated into some other library file such as SYSTEM.LIBRARY.

Separate compilation has several advantages in the development of any large or complicated program, because it allows you to approach the task as a group of smaller tasks which are linked together in a simple and logical way. Several of the powerful features of Apple Pascal are implemented as UNITs, as we will see in Chapter 7. To use a separately compiled UNIT, a program must contain a USES declaration with the name of the UNIT; the program is then called a host program.

There are two kinds of UNITs: Regular UNITs and Intrinsic UNITs. When a host program USES a Regular UNIT, the UNIT's code is inserted into the host program's codefile by the Linker. This need only be done once unless the UNIT is modified and recompiled; then it must be relinked into the host program.

When a host program USES an Intrinsic UNIT, the UNIT's code remains in its library file and is automatically loaded into memory when the host program is executed. This keeps the size of the host program's codefile down, which is particularly important if many programs use the UNIT. It also allows the UNIT to be modified and recompiled without the need to relink.

The Compiler's NOLOAD and RESIDENT options (see Chapter 4) allow further control over the handling of Intrinsic UNITs and SEGMENT procedures and functions. NOLOAD prevents any UNIT from being automatically loaded until its code is activated by the host program. The RESIDENT option can modify the effect of NOLOAD or of a SEGMENT procedure or function; it forces a procedure or function to be kept in memory over a specified range of program execution -- specifically, as long as the procedure or function containing the RESIDENT option is active, the procedure named in the RESIDENT option is kept in memory.

Finally, there is the EXTERNAL mechanism. This allows a procedure or function to be declared in a Pascal host program, without any statements except a heading and the word EXTERNAL. The procedure or function is implemented separately in assembly language, assembled, and then linked into the host program with the Linker. This can be advantageous for procedures or functions which must run very fast.

SEGMENT PROCEDURES AND FUNCTIONS

Declarations of SEGMENT procedures and functions are identical to ordinary Pascal procedures and functions except that the word PROCEDURE or FUNCTION is preceded by the word SEGMENT. For example:

```
SEGMENT PROCEDURE INITIALIZE;
BEGIN
  (* Pascal statements *)
END;

SEGMENT FUNCTION FFT(DOMAIN:MPTR): NPTR;
BEGIN
  (* Pascal statements *)
END;
```

Program behavior does not differ; however, the code and data for a SEGMENT procedure or function are in memory only while the procedure or function is actually running. This can be modified by use of the Compiler option (*$R name*) as explained in Chapter 4.

Any procedure or function definition may have the word SEGMENT. This includes FORWARD definitions and nested definitions.

The advantage of using SEGMENT procedures is the ability to fit large programs into the available memory. To write such a program, divide it into two or more main tasks which are implemented as SEGMENT procedures. To be effective, each SEGMENT should be substantial in size and the program should be designed so that SEGMENTs are not swapped in and out too frequently.

REQUIREMENTS AND LIMITATIONS

The disk which holds the code file for the program must be on line (and in the same drive as when the program was started) whenever one of the SEGMENT procedures is to be called. Otherwise, the system will attempt to retrieve and execute whatever information now occupies that particular location on the disk now in that drive, usually with very displeasing results.

SEGMENT procedures must be the first procedure declarations that contain code-generating statements.

LIBRARIES AND UNITS

So far, we have seen Pascal programs which are compiled into codefiles; a codefile can be R(un or eX(ecuted. Now we will consider UNITs, which are compiled into libraries. Two or more libraries can be combined into one file. A library is not R(un or eX(ecuted; instead, it is used by one or more programs.

A library contains code for procedures and/or functions which are available to any program that uses the library, just as if they were defined in the program itself. For example, the Apple Pascal System comes with a library called SYSTEM.LIBRARY which contains code for several UNITs; one of the UNITs is called TURTLEGRAPHICS, and it provides a set of procedures and functions for high-resolution graphics on the Apple. To use these procedures and functions, a program need only have the line

```
USES TURTLEGRAPHICS;
```

after the program heading. The program can then use a TURTLEGRAPHICS procedure such as TURNTO or MOVE.

You can create and compile your own UNITs, and either add them to SYSTEM.LIBRARY or build your own libraries by using the LIBRARY utility described in the Apple Pascal Operating System Reference Manual.

If a UNIT used by your program is contained in the SYSTEM.LIBRARY file, a R(un command will automatically invoke the Linker to do the necessary linking. Otherwise, you must explicitly invoke the Linker. Note that if the UNIT is not contained in the SYSTEM.LIBRARY file, you must use the (*$U filename*) option of the compiler to tell the compiler which library file contains the unit. The (*$U filename*) is placed anywhere before the appearance of the UNIT name in the USES declaration.

UNITS AND USES

The source text for a UNIT has a form somewhat similar to a Pascal program, as explained in detail further on. Briefly, it consists of four parts:

- A heading.
- An INTERFACE part which defines the way the host program communicates with the procedures and functions of the UNIT.
- An IMPLEMENTATION part which defines the procedures and functions themselves.
- An "initialization" which consists of a BEGIN and an END with any number of statements between them. This is the "main program" of the UNIT, and is automatically executed at the beginning of the host program. Note that the initialization

may consist of just the BEGIN and END, with no statements between them.

There are two different flavors of UNITs called Regular and Intrinsic.

Regular UNITs

The heading of a Regular UNIT has the form

```
UNIT name;
```

The UNIT is linked into the host program just once after the program is compiled, and the entire UNIT's code is actually inserted in the host program's codefile at that time.

Intrinsic UNITs

Intrinsic UNITs can only be used by installing them in the SYSTEM.LIBRARY file. This is done after compilation by using the LIBRARY utility program (see Apple Pascal Operating System Reference Manual).

An Intrinsic UNIT is "pre-linked," and its code is never actually inserted into the host program's codefile. When you R(un the host program, the Linker is not called and does not have to be on line. The Intrinsic UNIT's code is loaded into memory when the host program is to be executed. Thus an intrinsic UNIT can be used in many different programs, but there is only one stored copy of the UNIT's code.

This can be especially useful when writing for a one-drive system which does not have room for the Linker or for large programs on the main system diskette. Note that the SYSTEM.LIBRARY file must be on line each time the calling program is executed.

The heading of an Intrinsic UNIT has the form

```
UNIT name;
INTRINSIC CODE csegnum [DATA dsegnum];
```

where csegnum and dsegnum are the segment numbers to be associated with the UNIT in when it is installed in the SYSTEM.LIBRARY file. You choose these numbers, and the system uses them to identify the UNIT at run time. Segment numbers range from Ø to 31, but certain numbers between Ø and 15 must not be used (see below). The UNIT will generate a data segment if it declares any variables not contained in procedures or functions.

The code segment will be associated with segment csegnum and its data segment (if there is one) will be associated with segment dsegnum.

Every unit in a library has a specific segment number associated with it. The segment numbers used by items already in the library are shown in parentheses by the LIBRARY and LIBMAP utility programs (see Apple Pascal Operating System Reference Manual). In choosing segment numbers for an Intrinsic UNIT, the constraint is that when the host program runs, the segment numbers used by the program must not conflict. Observe the following:

- While any program is executing, the system uses segment Ø and the main program body uses segment 1. Therefore, never use these numbers for an Intrinsic UNIT.
- Segments 2 through 6 are reserved for use by the system.
- If the program declares any SEGMENT procedures or functions, these procedures or functions use sequentially increasing segment numbers starting at 7.
- Each UNIT used by the program uses the segment number shown in the library listing.
- If possible, avoid any duplication of segment numbers in the library.

Generally, it is a good idea to use segment numbers in the range from 16 through 31.

The compiler's SWAPPING option,

```
(*$S+*)
```

should always be used when a UNIT is compiled. It should precede the heading of the UNIT.

The INTERFACE Part of a UNIT

The first part of a UNIT is the INTERFACE.

The INTERFACE part immediately follows the UNIT's heading line. It declares constants, types, variables, procedures and functions that are public -- that is, the host program can access them just as if they had been declared in the host program. The INTERFACE portion is the only part of the UNIT that is "visible" from the outside; it specifies how a host program can communicate with the UNIT.

Procedures and functions declared in the INTERFACE are abbreviated to nothing but the procedure or function name and the parameter specifications, as shown in the example below.

The IMPLEMENTATION Part of a UNIT

The IMPLEMENTATION part immediately follows the last declaration in the INTERFACE.

The IMPLEMENTATION begins by declaring those labels, constants, types, variables, procedures and functions that are private -- that is, not accessible to the host program. Then the public procedures and functions that were declared in the INTERFACE are defined. As shown in the example below, they are defined without parameters or function result types, since these have already been defined in the INTERFACE.

The Initialization Part of a UNIT

At the end of the IMPLEMENTATION part, following the last function or procedure, there is the "initialization" part. This is a sequence of statements preceded by BEGIN and terminated with END. The resulting code runs automatically when the host program is executed, before the host program is run. It can be used to make any preparations that may be needed before the procedures and functions of the UNIT can be used. For example, the initialization part of the TRANSCEND UNIT in SYSTEM.LIBRARY generates a table of trigonometric values to be used by the transcendental functions. If you don't want any initialization to take place, you must still have the END followed by a period.

AN EXAMPLE UNIT

Let's sketch out an imaginary Intrinsic UNIT that needs a DATA segment, to demonstrate the information given above.

```
(*$S+*)              (* Swapping is required for compiling UNITs *)

UNIT FROG; INTRINSIC CODE 25 DATA 26;

    INTERFACE                                  (* This stuff is public *)
        CONST FLYSIZE = 1Ø;
        TYPE WARTTYPE = (GREEN,BROWN);
        VAR FROGNAME:STRING[2Ø];          (* Will need Data segment *)
        PROCEDURE JUMP(DIST:INTEGER);
        FUNCTION WARTS:INTEGER;


    IMPLEMENTATION                             (* This stuff is private *)
        CONST PI = 3.14159;
        TYPE ETC = Ø..13;
        VAR FROGLOC:INTEGER;

        PROCEDURE JUMP;                  (* Note: no parameters here *)
          BEGIN
            FROGLOC := FROGLOC + DIST
          END;

        FUNCTION WARTS;
          BEGIN
            (* Function definition here *)
          END;

        (* More procedures and functions here *)


    BEGIN
        (* Initialization code, if any, goes here *)
    END.
```

Variables of type FILE must be declared in the INTERFACE part of a UNIT. A FILE declared in the IMPLEMENTATION part will cause a syntax error upon compilation.

USING THE EXAMPLE UNIT

The UNIT above, properly completed, would then be compiled. Then the UNIT would be installed in SYSTEM.LIBRARY, using the LIBRARY utility. Once in the library, the UNIT could then be used by any Pascal host program. A sample program to use our UNIT is sketched out below:

```
PROGRAM JUMPER;

    USES FROG;
    CONST ... ;
    TYPE ... ;
    VAR ... ;
    PROCEDURE ... ;
    FUNCTION ... ;

    BEGIN
      ... ;
      ... ;
      ... ;
    END.
```

A program must indicate the UNITs that it USES before the declaration part of the program; procedures and functions may not contain their own USE declarations. At the occurrence of a USES declaration, the Compiler references the INTERFACE part of the UNIT as though it were part of the host text itself. Therefore all constants, types, variables, functions, and procedures publicly defined in the UNIT are global. Name conflicts may arise if the user defines an identifier that has already been publicly declared by the UNIT. If the UNIT is not in the SYSTEM.LIBRARY, the USES declaration must be preceded by a "use library" option to tell the compiler what library file contains the UNIT.

NESTING UNITS

A UNIT may also USE another UNIT, in which case the USES declaration must appear at the beginning of the INTERFACE part. For example, our UNIT FROG might use the graphics UNIT TURTLEGRAPHICS:

```
(*$S+*)
UNIT FROG; INTRINSIC CODE 25 DATA 26;

     INTERFACE
         USES TURTLEGRAPHICS;
         CONST FLYSIZE = 1Ø;
           . . . . .
              etc.
```

When you later use such a UNIT, your host program must declare that it USES the deepest nested UNIT first:

```
PROGRAM JUMPER;

    USES TURTLEGRAPHICS,FROG;
```

There is one limitation: an Intrinsic UNIT cannot USE a Regular UNIT.

CHANGING A UNIT OR ITS HOST PROGRAM

For test purposes, you may define a Regular UNIT right in the host program, after the heading of the host program. In this case, you will compile both the UNIT and the host program together. Any subsequent changes in the UNIT or host program require that you recompile both.

Normally, you will define and compile a Regular UNIT separately and use it as a library file (or store it in another library by using the LIBRARY utility). After compiling a host program that uses such a UNIT, you must link that UNIT into the host program's codefile by executing the Linker. Trying to R(un an unlinked code file will cause the Linker to run automatically, looking for the UNIT in the system library. Trying to X(ecute an unlinked file causes the system to remind you to link the file.

Changes in the host program require that you recompile the host program. You must also link in the UNIT again, if it is not Intrinsic.

Changes in a Regular UNIT require you to recompile the UNIT, and then to recompile and relink all host programs (or other UNITs) which use that UNIT.

INTRINSIC UNITs and their host programs can be changed as described above, but they do not have to be relinked.

EXTERNAL PROCEDURES AND FUNCTIONS

EXTERNAL procedures (.PROC's) are separately assembled assembly-language procedures, often stored in a library file. Host programs that require EXTERNAL procedures must have them linked into the compiled code file.

A host program declares that a procedure (or function) is EXTERNAL in much the same way as a procedure is declared FORWARD. A standard heading is provided, followed by the keyword EXTERNAL:

```
PROCEDURE FRAMMIS (WIDGET, GIDIBRION:INTEGER);
EXTERNAL;
```

There is one special rule for the heading of an EXTERNAL procedure or function: A VAR parameter can be declared without any type.

Calls to the EXTERNAL procedure use standard Pascal syntax, and the Compiler checks that calls to the EXTERNAL agree in type and number of parameters with the EXTERNAL declaration. It is the user's responsibility to ensure that the assembly-language procedure respects the Pascal EXTERNAL declaration. The Linker checks only that the number of words of parameters agree between the Pascal and assembly-language declarations. For more information see the Apple Pascal Operating System Reference Manual.

The conventions of the surrounding system concerning register use and calling sequences must be respected by writers of assembly-language routines. On the Apple, all registers are available, and zero-page hexadecimal locations Ø through 35 are available as temporary variables. However, the Apple Pascal system also uses these locations as temporaries, so you should not expect data left there to be there when you execute the routine the next time. You can save variables in non-zero page memory by using the .BYTE or .WORD directives in your program to reserve space.

For assembly language functions (.FUNC's) the sequence is essentially the same, except that:

- Two words of zeros are pushed by the Compiler after any parameters are put on the stack.
- After the stack has been completely cleaned up at the routine exit time, the .FUNC must push the function result on the stack.

For an example of an EXTERNAL assembly-language procedure and an EXTERNAL assembly-language function, called from a Pascal program, see the example in the Apple Pascal Operating System Reference Manual. The EXTERNAL routine in that example is manually Linked into the calling program.

CHAPTER 6
OTHER DIFFERENCES

IDENTIFIERS

The underscore character _ is allowed in identifiers; however, the compiler ignores it. Therefore the identifiers

```
FIG_LEAF
FIGLEAF
```

are equivalent. (The Apple keyboard does not have the underscore character, but some external terminals do.)

CASE STATEMENTS

In Standard Pascal, if there is no case label equal to the value of the case selector, the result of the case statement is undefined. In Apple Pascal, if there is no case label matching the value of the case selector, then the next statement executed is the statement following the case statement.

COMMENTS

The Apple Pascal compiler recognizes any text appearing between either the symbols (* and *) or the symbols { and } as a comment. Text appearing between these symbols is ignored by the Compiler unless the first character of the comment is a dollarsign, in which case the comment is interpreted as a compiler option (see Chapter 4).

If the beginning of the comment is delimited by the (* symbol, the end of the comment must be delimited by the matching *) symbol, rather than the } symbol. When the comment begins with the { symbol, the comment continues until the matching } symbol appears. This feature allows you to "comment out" a section of a program which itself contains comments. This applies to external terminals only, since the only comment delimiter available on the Apple is the pair (* and *). An example of how the two kinds of comment delimiters are used on an external terminal:

```
{     XCP := XCP + 1;  (* ADJUST FOR SPECIAL CASE... *)   }
```

The compiler does not keep track of nested comments. When a comment symbol is encountered, the text is scanned for the matching comment symbol. The following text will result in a syntax error:

```
(* THIS IS A COMMENT  (* NESTED COMMENT *)  END OF FIRST COMMENT *)
                                              ^error here.
```


GOTO

Apple Pascal has a more limited form of GOTO statement than Standard Pascal. The destination of the GOTO statement must be in the same procedure as the GOTO statement itself (considering the main program to be a procedure).

The compiler considers a GOTO statement to be illegal unless the compiler option (*$G+*) is used; see Chapter 4.

PROGRAM HEADINGS

Although the Apple Pascal Compiler will permit a list of file parameters to be present following the program identifier (as in Standard Pascal), these parameters are ignored by the compiler and have no affect on the program being compiled.

SIZE LIMITS

The following is a list of maximum size limitations imposed upon the user by the current implementation of Apple Pascal:

- Maximum number of bytes of object code in a PROCEDURE or FUNCTION is 12ØØ. Local variables in a PROCEDURE or FUNCTION can occupy a maximum of about 18ØØØ words of memory.
- Maximum number of characters in a STRING variable is 255.
- Maximum number of elements in a SET is 32 * 16=512.
- Maximum number of segments a program can use is 16. This includes one segment for the main program, one for each SEGMENT PROCEDURE and SEGMENT FUNCTION declared in the program, and one for each Regular UNIT that the program USES.
- Maximum number of PROCEDUREs or FUNCTIONs within a segment is 149.
- Maximum integer that can be represented is 32767; minimum is -32768.
- Maximum precision for REAL values is 32 bits.

EXTENDED COMPARISONS

Apple Pascal allows = and <> comparisons of arrays of exactly the same type and of record structures of exactly the same type. This can be

done without subscripting (in the case of arrays) or field identifiers (in the case of records). For example, given the declarations

```
VAR A: ARRAY[Ø..1Ø] OF INTEGER;
    B: ARRAY[Ø..1Ø] OF INTEGER;
```

then the following statement is legal:

```
IF A=B THEN ...
```

and the statement following the THEN will be executed if each element of A is equal to the corresponding element of B.

PROCEDURES AND FUNCTIONS AS PARAMETERS

Apple Pascal does not allow a PROCEDURE or FUNCTION to be declared as a formal parameter in the parameter list of another PROCEDURE or FUNCTION.

RECORD TYPES

There are two restrictions on record type declarations which are different from Standard Pascal syntax:

- A null field list is illegal; in other words the construction

  ```
  RECORD END;
  ```

  will cause an error.

- A null field within the parentheses of a variant field list is illegal; in other words a semicolon just before the closing parenthesis will cause an error.

THE ORD FUNCTION

The ORD function will accept a parameter of type POINTER, and return the numerical value of the pointer.

When the ORD function is given a BOOLEAN value as an actual parameter, the result is not always Ø or 1. It is most unlikely that a working program will ever encounter this situation, since there is little reason to take the ORD of a BOOLEAN value.

CHAPTER 7
SPECIAL UNITS SUPPLIED FOR THE APPLE

APPLE GRAPHICS: THE TURTLEGRAPHICS UNIT

This graphics package is called "Turtlegraphics" since it is based on the "turtles" devised by S. Papert and his coworkers at the Massachusetts Institute of Technology. To make graphics easy for children who might have difficulty understanding Cartesian coordinates, Papert et al. invented the idea of a "turtle" who could walk a given distance and turn through a specified angle while dragging a pen along. Very simple algorithms in this system (which could be called "relative polar coordinates") can give more interesting images than an algorithm of the same length in Cartesian coordinates.

Before any graphics can be used, they must be enabled by placing this declaration immediately after the program heading:

```
USES TURTLEGRAPHICS;
```

If this declaration appears, the graphics procedures and functions described in this section can be used. This declaration tells the Pascal system to get the graphics programs from the library. The SYSTEM.LIBRARY file must be on line when the program is R(un or eX(ecuted.

THE APPLE SCREEN

The Apple screen is a rectangle, with the origin (X=Ø,Y=Ø) at the LOWER LEFT corner. The upper right corner has the coordinates (X=279,Y=191). Since points may only be placed at integral coordinates, all arguments to the graphics functions are INTEGERs. (You can supply a REAL argument; it will be rounded to an INTEGER.)

There are two different screen images stored in the Apple's memory. One of them holds the text that you see when the computer is first turned on. The other holds a graphic image. There are three procedures that switch between the modes. They are INITTURTLE, TEXTMODE and GRAFMODE.

THE INITTURTLE PROCEDURE

This procedure has no parameters. It clears the screen, and allows the screen to be used for graphics rather than text. It is a good idea to use this routine before starting any graphics.

INITTURTLE does a few other things as well: the turtle (more about it later) is placed in the center of the screen facing right, the pen color is set to NONE (more about this later too) and the viewport is set to full screen (read on).

THE GRAFMODE PROCEDURE

This procedure has no parameters. It switches the monitor or TV to show the graphics screen, without the other initialization that INITTURTLE does. It is usually used to show graphics in a program that switches between graphics and text display.

THE TEXTMODE PROCEDURE

This procedure has no parameters. It switches from graphics mode (obtained by INITTURTLE or GRAFMODE) to showing text. When you switch to text mode, the image that you saw in GRAFMODE is not lost, but will still be there when you use GRAFMODE to go into graphics mode again (unless you deliberately changed it.) Upon termination of any program that uses graphics, the system automatically goes back into text mode.

THE VIEWPORT PROCEDURE

This procedure has the form

```
VIEWPORT (LEFT, RIGHT, BOTTOM, TOP)
```

where the four parameters give the boundaries you want the VIEWPORT to have. If you don't use this procedure, Apple Pascal assumes that you want to use the whole screen for your graphics.

There are occasions when it is handy to use only part of the screen, while safeguarding the rest from accidental use. For example, a small square near the middle of the screen might be selected as a viewport by the statement:

```
VIEWPORT (13Ø, 15Ø, 86, 1Ø6)
```

This example would allow the screen-plotting of all points whose X-coordinates are from 13Ø through 15Ø and whose Y-coordinates are from 86 through 1Ø6.

When a line is drawn using any of the graphic commands, it is automatically clipped so that only the portion which lies within the current viewport is displayed. Points whose coordinates are not in the current viewport, even those points that would not be on the screen at all, are legal but are ignored.

This allows some dramatic effects. It also allows you to plot off-screen all day, and never see a thing or get an error message. Clipping cannot be disabled.

USING COLOR: PENCOLOR

The PENCOLOR procedure sets the pen color. It has the form

```
PENCOLOR (COLOR)
```

The simplest colors are

```
WHITE

WHITE1 (two dots wide, for use with green and violet)

WHITE2 (two dots wide, for use with orange and blue)

BLACK

BLACK1 (two dots wide, for use with green and violet)

BLACK2 (two dots wide, for use with orange and blue)

GREEN

VIOLET

ORANGE

BLUE
```

If you'd like the drawing to be in GREEN, use the statement:

```
PENCOLOR (GREEN)
```

It may seem strange that aside from WHITE, BLACK, GREEN, VIOLET, ORANGE, and BLUE, there are two additional flavors of WHITE and BLACK. These are due to the intricate (not to say bizarre) way that color televisions produce their color, interacting with the technique that Apple uses to get a lot of color very economically. Rather than explaining how this all works, suffice it to say here that WHITE and BLACK give the finest lines possible, and the colors give a wider line in order to make the colors show. If you wish to make a white or black line that corresponds exactly in position and width with a green or violet line then you should use WHITE1 or BLACK1. If you wish to make a white or black line that corresponds exactly in position and width with an orange or blue line, then you should use WHITE2 or BLACK2.

On a black-and-white monitor or TV set, just use WHITE and BLACK.

The remaining colors are not really colors at all but are equally useful:

- NONE: Drawing with this "color" produces no change on the screen. It is useful for moving the turtle without drawing a line.

- REVERSE: Drawing with REVERSE changes BLACK to WHITE and WHITE to BLACK. It also changes WHITE1 to BLACK1, WHITE2 to BLACK2, GREEN to VIOLET and ORANGE to BLUE and vice versa. It is rather a magical "color". It allows you to draw, say, a line across a complex background and have it still show up.

- RADAR: This "color" has been left unused for future applications.

MORE COLOR: FILLSCREEN

The FILLSCREEN procedure has the form

```
FILLSCREEN (COLOR)
```

FILLSCREEN fills the entire viewport with the specified color. For example

```
FILLSCREEN (BLACK)
```

clears the viewport. The statement

```
FILLSCREEN (REVERSE)
```

makes a "negative" of the contents of the viewport.

When you invoke TURTLEGRAPHICS, a new variable type called SCREENCOLOR is automatically created. It is defined as follows:

```
SCREENCOLOR = (NONE, WHITE, BLACK, REVERSE, RADAR, BLACK1, GREEN,
               VIOLET, WHITE1, BLACK2, ORANGE, BLUE, WHITE2);
```

SCREENCOLOR has all the usual characteristics of a Pascal type. It is useful when you declare a variable that will be used to store a color.

TURTLE GRAPHIC PROCEDURES: TURNTO, TURN, AND MOVE

At last we're back to the imaginary turtle. Initially, the turtle sits at the center of the screen, facing right. The turtle can only do two things: it can turn or it can walk in the direction it is facing. As it walks, it leaves behind a trail of ink (!) in the current pen color.

The TURNTO procedure has the form

```
TURNTO (DEGREES)
```

where DEGREES is an integer which is treated modulo 360; thus its effective value is in the range -359 through 359. When invoked, this procedure causes the turtle to turn from its present angle to the indicated angle. 0 is exactly to the right, and counterclockwise rotation represents increasing angles. This command never causes any change to the image on the screen.

The TURN procedure has the form

```
TURN (DEGREES)
```

where DEGREES is again an integer which is treated modulo 360; thus its effective value is in the range -359 through 359. This procedure causes the turtle to rotate counterclockwise from its current direction through the specified angle. It causes no change to the image on the screen.

The MOVE procedure has the form

```
MOVE (DISTANCE)
```

where DISTANCE is an integer. This procedure makes the turtle move IN THE DIRECTION IN WHICH IT IS POINTING a distance given by the integer DISTANCE. It leaves a trail in the current pen color. The sequence of statements:

```
PENCOLOR (WHITE);
MOVE (50);
TURN (120);
MOVE (50);
TURN (120);
MOVE (50)
```

draws an equilateral triangle, for instance.

TURTLE GRAPHIC FUNCTIONS: TURTLEX, TURTLEY, TURTLEANG, AND SCREENBIT

These functions allow you to interrogate the computer about the current state of the turtle and the screen.

The TURTLEX and TURTLEY functions (no parameters) return integers giving the current X and Y coordinates of the turtle.

The TURTLEANG function (no parameters) returns an integer giving the current turtle angle as a positive number of degrees. Note that if you use TURNTO and then TURTLEANG, the value returned by TURTLEANG may not be the same value you gave with TURNTO. For example, after

```
TURNTO(-90)
```

TURTLEANG will return 270, not -90.

The SCREENBIT function has the form

```
SCREENBIT (X,Y)
```

where X and Y are screen coordinates. This function returns the BOOLEAN value TRUE if the specified location on the screen is not black, and FALSE if it is black. It doesn't tell you what color is at that point, but only whether there is something non-black there or not.

CARTESIAN GRAPHICS: THE MOVETO PROCEDURE

Earlier we said that in turtle graphics, the turtle can only walk in the direction it is facing. But in Cartesian graphics, the turtle can move to a specified point on the screen without turning. The MOVETO procedure has the form

```
MOVETO (X, Y)
```

where X and Y are screen coordinates. MOVETO moves the turtle to the point (X,Y). This creates a line in the current pen color from the turtle's last position to the point (X,Y).

The direction of the turtle is not changed by MOVETO.

GRAPHIC ARRAYS: THE DRAWBLOCK PROCEDURE

The DRAWBLOCK procedure has the form

```
DRAWBLOCK (SOURCE, ROWSIZE, XSKIP, YSKIP, WIDTH, HEIGHT,
           XSCREEN, YSCREEN, MODE)
```

where the SOURCE parameter is the name (without subscripts) of a variable which should be a two-dimensional PACKED ARRAY OF BOOLEAN (see note below). All the other parameters are integers.

DRAWBLOCK treats each BOOLEAN element of SOURCE as a "dot" -- true for white or false for black. It copies the array of "dots" (or a portion of it) from memory onto the screen to form a screen image. The first dimension of the array is the number of rows in the array; the second dimension is the number of elements in each row.

You may choose to copy the entire SOURCE array, or you may choose to copy any specified "window" from the array, using only those dots in the array from XSKIP to XSKIP+WIDTH and from YSKIP to YSKIP+HEIGHT. Furthermore, you can specify the starting screen position for the copy, at (XSCREEN, YSCREEN).

- SOURCE is the name of the two-dimensional PACKED ARRAY OF BOOLEAN to be copied (see note below).

- SIZE is the number of bytes (not dots) per row in the array. You can calculate this from the formula

  ```
  2*( (X+15) DIV 16 )
  ```

  where X is the number of dots in each row.

- XSKIP tells how many horizontal dots in the array to skip over before the copying process is started.

- YSKIP tells how many vertical dots in the array to skip over before beginning the copying process. Note that copies are made starting from the bottom up -- i.e. the first row copied from the array is the bottom row of the screen copy.

- WIDTH tells how many dots' width of the array, starting at XSKIP, will be used.

- HEIGHT tells how many dots' height of the array, starting at YSKIP, will be used.

- XSCREEN and YSCREEN are the coordinates of the lower left corner of the area to be copied into. The WIDTH and HEIGHT determine the size of the rectangle.

- MODE ranges from Ø through 15. The MODE determines what appears on the portion of the screen specified by the other parameters. It is quite a powerful option, which can simply send white or black to the screen, irrespective of what is in the array, copy the array literally, or combine the contents of the array and the screen and send the result to the screen. The following table specifies what operation is performed on the data in the array and on the screen, and thus what appears on the screen. (The logical notation uses A for the array, and S for the screen. The symbol ~ means NOT.)

MODE	Effect
Ø	Fills area on screen with black.
1	NOR of array with screen. (A NOR S)
2	AND of array with complement of screen. (A AND ~S)
3	Complements area on screen. (~S)
4	AND of complement of array with screen. (~A AND S)
5	Complements the array. (~A)
6	XOR of array with screen. (A XOR S)
7	NAND of array with screen. (A NAND S)
8	AND of array with screen. (A AND S)
9	EQUIVALENCE of array with screen. (A = S)
1Ø	Copies array to screen. (A)
11	OR of array with complement of screen. (A OR ~S)
12	Screen replaces screen. (S)
13	OR of complement of array with screen. (~A OR S)
14	OR of array with screen. (A OR S)
15	Fills area on screen with white.

The demonstration program GRAFDEMO.TEXT, on APPLE3:, contains many examples of how to use the turtlegraphics routines. In particular, procedures BUTTER1, etc., give strings to procedure STUFF, which converts them to a PACKED ARRAY OF BOOLEAN named BUTTER; and procedure FLUTTER uses the DRAWBLOCK routine to display the array BUTTER on the screen.

Actually, the SOURCE parameter can be of any type except a FILE type; DRAWBLOCK really deals with an array of bits in memory which begins at the address of SOURCE and whose size and organization depend on the other parameters. For example, the following procedure uses a single BOOLEAN variable instead of an array. The procedure plots a single dot on the screen at specified coordinates (X,Y):

```
PROCEDURE PLOTDOT(X, Y: INTEGER);
  VAR DOT:BOOLEAN;
  BEGIN
    DRAWBLOCK(DOT,1,Ø,Ø,1,1,X,Y,3)
  END;
```