Apple III
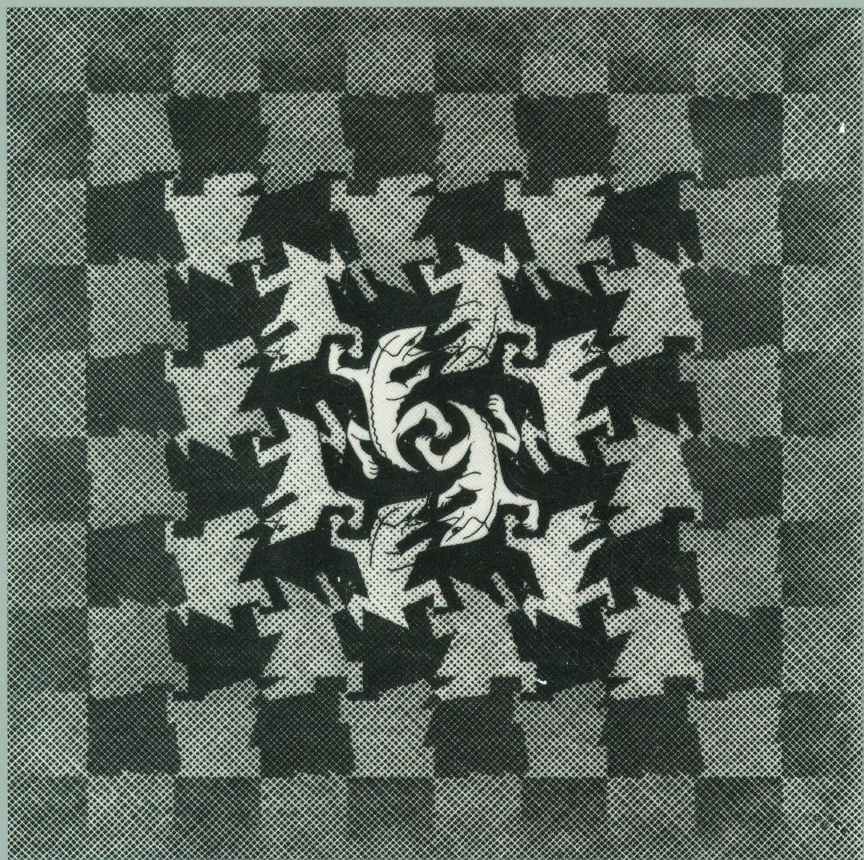
Pascal
Programmer's Manual   Volume 1

Notice

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

Disclaimer of All Warranties And Liabilities

Apple III Pascal

Programmer's Manual

Volume 1

Acknowledgements

Contents

Volume I—Chapters

B The PGRAF Unit   11

C The CHAINSTUFF Unit   39

D The APPLESTUFF Unit   45

E *Floating-Point Arithmetic*   55

F *The Apple III Pascal Compiler*   87

G *Special Techniques*   113

H Comparison To Apple II Pascal   *127*

I Syntax Diagrams   *133*

J *Tables*   153

Preface

The Apple III Pascal system is described in three manuals:

    Apple III Pascal: Intoduction, Filer, and Editor
    Apple III Pascal Program Preparation Tools
    Apple III Pascal Programmer's Manual (Volumes 1 and 2)

Before using the Apple III Pascal system or reading its manuals, you should be familiar with starting up the Apple III as described in the Apple III Owner's Guide.

When you are familiar with the contents of that manual, begin reading the Apple III Pascal: Introduction, Filer, and Editor manual. The Filer and the Editor described in this manual are needed by everyone who uses the Pascal system. If you are familiar with the Apple II Pascal system, this manual will show you the differences in operation between the two systems.

Apple III Pascal Program Preparation Tools is the next manual that you should read before you start to develop Pascal and assembly-language programs to run on the Apple III. The components of the Apple III Pascal system covered in this manual include

- The Linker, used to combine separately developed program segments stored in libraries with your application program.

- The Apple III Pascal 6502 Assembler, used to translate assembly-language source files produced by the Pascal Editor into machine-language code files.

- The Librarian, used to put commonly used routines into libraries for use with application programs.

Your main source of information while developing Pascal programs
will be the two volumes of the Apple III Pascal Programmer's
Manual, which contain a complete description of the Pascal
language on the Apple III and the use of the Apple III Pascal
Compiler.

The Contents of This Manual

This manual describes the complete Apple III Pascal language.
Except for the introductory material in Chapters 1 and 2, this is
an explanatory reference manual rather than a textbook; it does
not assume that you know anything about Pascal, but it does
assume that you are familiar with computer programming in some
language.

Please note that a large and detailed index is provided at the
end of this manual; you will probably need it when you are using
the manual for reference purposes. The index does not point to
every occurrence of a word or phrase in the manual; instead it
points to the pages that have significant information about the
topic associated with the word or phrase.

Volume 1 of this manual contains the chapters; Volume 2 contains
the appendices and the index. Here is a brief description of the
contents:

- Chapter 1 is an introduction to the Pascal language,
  comparing it with other well-known languages and giving
  a very simple program as an example.

- Chapter 2 is an extensive overview of Pascal. Every
  major concept and construction in the language is
  introduced here at an intuitive level.

- Chapters 3 through 11 provide complete, detailed
  information about every major feature of the language.

- Chapters 12 through 15 provide complete, detailed
  information about the more specialized features of the
  language. These features are needed for certain large
  or specialized programs.

- Appendices A through E describe the standard library
  facilities of Apple III Pascal. These are sets of
  procedures and functions for special purposes such as
  graphics, audio, joystick inputs, and special
  arithmetic features.

- Appendix F is a complete reference manual for the Apple
  III Pascal Compiler, including details of operation and
  all of the Compiler options.

- Appendices G through J are supplementary information on
  various topics. In particular, Appendix J is a
  collection of useful tables.

- Appendix K provides information on the use of Apple II
  TURTLEGRAPHICS on the Apple III.

Two special symbols are used throughout this manual to draw your
attention to particular items of information.

The pointing hand indicates something particularly
interesting or useful.

The eye is used for points you need to be cautious about.

Syntax Diagrams

Throughout this manual, the syntax of the Pascal language is
indicated by means of syntax diagrams, also known as "railroad
tracks." These diagrams are easy to follow once you are used to
them: begin at the upper left and follow the arrows. Every
possible path through the diagram represents a valid construction
in Pascal. For example:

while statement

This diagram tells us that a "while statement" consists of the
word WHILE, followed by an expression, followed by the word DO,
followed by a statement.

The words WHILE and DO are enclosed in rounded "bubbles;" this
means that they are reserved words or symbols of the language, to
be typed as shown. The words expression and statement are in
boxes with square corners; this means that they are higher-level
constructions, which have their own syntax diagrams.

Here is an example where there is more than one path through the
diagram:

identifier
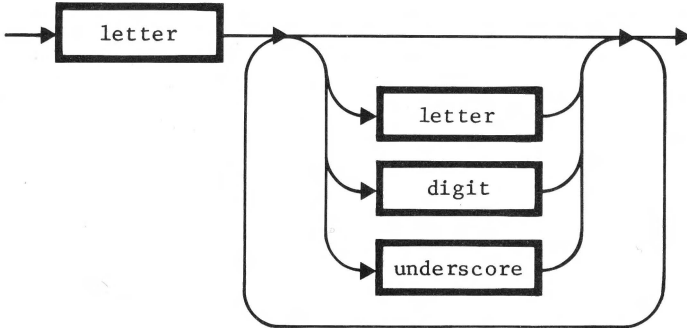
This tells us that an identifier begins with a letter, and this
letter may be followed by a letter, a digit, an underscore, or
nothing. From here, there is the possibility of looping back to
add another letter, digit, underscore, or nothing. This can be
repeated indefinitely (in principle), so the syntax says that an
identifier can be of any length. In practice, of course, there
is a limit which the syntax does not show.

Note that Appendix I contains a full set of syntax
diagrams.

Syntax of Procedure and Function Calls

Pascal provides a number of built-in procedures and functions
which are activated by means of "calls." Most of these use a
simple kind of syntax in which there is only one path through the
diagram, and in these cases a diagram is not shown. Instead, a
"form" is given; for example, the form of the REWRITE procedure
is

        REWRITE (FILEID, PATHNAME)

The word REWRITE is the name of the procedure, and is to be typed
as shown; all words in parentheses are names for "parameters," to
be replaced with actual expressions or variable identifiers as
explained in the text. In this example, FILEID is to be replaced
by the identifier of a "file variable" and PATHNAME is to be
replaced by a string of characters that is the pathname of a
file.

A few procedures have a more complex form of syntax, and syntax
diagrams are used for these.

1

What is Apple III Pascal?

Introduction

Apple III Pascal is an implementation of the Pascal language for the Apple III computer. It is based on UCSD Pascal, which in turn is based on the original definition of Pascal by Kathleen Jensen and Niklaus Wirth in the Pascal User Manual and Report (Springer-Verlag, 1974). Chapter 2 is a comprehensive overview of the language. This chapter is a brief description to give the flavor of Pascal, especially in comparison to two other popular general-purpose languages, BASIC and FORTRAN.

Pascal is a modern high-level programming language that belongs to the family of Algol-like languages; that is, it is descended from the Algol language which introduced many of the fundamental ideas of modern high-level programming. Like Algol, Pascal is written free-form and is block-structured. It goes beyond Algol in several ways; the most important is that Pascal has a great variety of different data types and allows the programmer to define new data types.

An Apple III Pascal program is first written as a text file, using the system's Editor; this text is called the source of the program. Next it is compiled by the Pascal Compiler. The Compiler produces a file of P-code, a special code that resembles machine language.

The P-code can then be "executed" by an interpreter program, which interprets the P-code instructions and executes them immediately. The interpreter program is known as the P-machine, and is an integral part of the software system. The use of P-code and an interpreter provides an important benefit: with a few limitations, the P-code of an Apple III Pascal program can also be executed by a different P-machine on a different computer.

An important part of the philosophy of high-level languages is to separate the programmer from the details of the computer hardware. In Pascal, for instance, there is no need to refer to specific physical addresses in memory. The system takes care of all memory management, without any intervention from the Pascal program. The system also provides control of all special machine features via Pascal statements; in most cases, this eliminates the need for writing machine-language routines for detailed

control of the hardware. However, there is a way to link
machine-language routines into a Pascal program.

Pascal vs. BASIC

If you are a BASIC programmer, you will find that Pascal is
different in some very fundamental ways:

- LINE NUMBERS: Pascal has no line numbers. In fact,
  line breaks mean nothing in a Pascal program. You can
  break up a statement into several lines or put several
  statements on one line. Statements are separated from
  each other by semicolons. You will find that the
  mechanics of writing, editing, or modifying a Pascal
  program are easier than with BASIC because you don't
  need to maintain line numbers.

- VARIABLES: All variables in a Pascal program must be
  declared before they can be used. A variable
  declaration associates an identifier (variable name)
  with one of the many data types of Pascal. (In BASIC,
  only arrays need to be declared, via the DIM
  statement.)

- FLOW OF CONTROL: Pascal has several methods for
  controlling the sequence in which statements are
  processed. These methods go beyond the IF, FOR, GOTO,
  and GOSUB/RETURN of BASIC. As a result, most Pascal
  programs are easier to read and understand than
  comparable BASIC programs. Pascal has a GOTO
  statement, but it is much less important than the other
  methods.

- PROCEDURES: In Pascal you can write a procedure, which
  is simply a subprogram. The main program can execute
  the subprogram by mentioning its name. This replaces
  the GOSUB/RETURN mechanism of BASIC and is more
  powerful, since the main program can supply parameter
  values to the procedure when it executes it.

- FUNCTIONS: A Pascal function is just a procedure that
  returns a value, in the same way as a BASIC
  user-defined function. A Pascal function definition
  can contain any number of statements, where a BASIC

user-defined function is usually severely limited in
the number of statements it can contain.

- BLOCK STRUCTURE: Block structure means that a
  procedure or function can have its own variables which
  are independent of the main program. A procedure or
  function can even have a variable of its own which has
  the same name as a variable in the main program.
  Because of this, a Pascal programmer can use a kind of
  discipline that is not possible in BASIC; the result is
  cleaner, more comprehensible programs.

- PHYSICAL ADDRESSES: There are no POKE, PEEK, or CALL
  statements in Pascal. A Pascal program doesn't use
  physical addresses; various mechanisms in Pascal make
  them unnecessary.

Pascal vs. FORTRAN

If you are a FORTRAN programmer, these differences will be
particularly noticeable:

- There are no line numbers.

- Identifiers (names) are less restricted.

- The format of the program text is less restricted.

- The program control structures are more constrained.

- Procedures and functions can be recursive: that is,
  they can call themselves either directly or
  indirectly.

- Block structure (see above) allows better program
  organization and eliminates the need for common
  blocks. Subprograms (procedures and functions) are
  written within the main program, and automatically have
  access to the main program's data.

- There is no equivalencing of variables.

- Variables can be created dynamically as the program
  runs, and referenced through pointers.

- There is no implicit typing. The type of every
  variable is explicitly declared.

- There are no OWN (SAVE) variables.

- I/O formatting facilities are simpler.

Structure of a Pascal Program

Every Pascal program has the following outline (words in capital
letters are "reserved words" of Pascal):

    program heading,
      containing the word PROGRAM and the program's name;

    declarations of any user-defined data types,
      variables, etc.

    definitions of any procedures and functions

    the word BEGIN

      any number of statements, separated by semicolons

    the word END, followed by a period

A Sample Program

Here is a very simple Pascal program, which displays 11 lines of
text on the screen. Each line displayed contains a different
number, counting from 0 to 10. The program is presented here
without explanation, just to show you what Pascal source text
looks like. The words between starred parentheses are
Pascal-language comments. At the end of the next chapter, we
will see this program again with an explanation.

```
PROGRAM FIRSTEXAMPLE;                    { program heading }

VAR I:INTEGER;                           { declaration of
                                              a variable }

PROCEDURE DISPLAY (J:INTEGER);      { procedure definition }
  BEGIN
    WRITELN;
    WRITELN('The number is ', J)
  END;                                   { end procedure
                                              definition }

BEGIN                                { begin program body }
  FOR I:=0 TO 10 DO DISPLAY(I)             { statement }
END.                                  { end of program }
```

The output from this program appears on the screen as follows:

```
The number is 0

The number is 1

The number is 2

The number is 3

The number is 4

The number is 5

The number is 6

The number is 7

The number is 8

The number is 9

The number is 10
```

2

Overview of Pascal

This chapter gives a brief account of each major feature of Apple III Pascal, and tells you where to find further information in this manual.

Pascal Source Text

The source text of a Pascal program is a sequence of <u>symbols</u>. Symbols are like the words, spaces, and punctuation marks that make up a paragraph of English sentences. The kinds of symbols that make up Pascal programs are

- Reserved words (the fixed vocabulary of Pascal)

- Identifiers (names: some made up by the programmer, and others built into the language)

- Numeric constants (numbers written in the program to be used as data)

- Character and string constants (characters written in the program to be used as data)

- Delimiters (special characters and punctuation)

These different kinds of symbols are described further under "Elements of the Language" below.

A Pascal program can also contain comments. A comment is text that is totally ignored by the Compiler; it serves to make the program more comprehensible to a human reader. Anything enclosed within the special symbols { and } is a comment; also, anything enclosed within the special symbols (* and *) is a comment. For example:

        {This is a comment}
        (*This is another comment*)

A comment formed with the {} delimiters can include the (* *) delimiters, and vice versa.

The source text of a Pascal program is written free-form: that is, you can break the text into lines in any way you like, you can indent the lines in any way you like, and you can insert spaces freely between symbols. For example, the Pascal statement

```
FOR I:= Ø TO N DO BEGIN A:=I;  B:=2*I+1 END
```

can also be written as follows:

```
    FOR I := Ø TO N DO BEGIN
                   A := I;
                   B := 2*I + 1
                END
```

or in many other ways. All that matters is the sequence of
symbols. It is customary to break the text into lines which are
meaningful, and to use indentation (as above) to improve
clarity. There are no specific rules for this; as you learn the
language you will understand the way indentation is used in the
examples and develop your own style.

Another way to write our sample Pascal statement is

```
    For i := Ø to n do Begin a:=i; b:=2*i + 1 End
```

In this manual, most Pascal program text is shown in capitals,
simply to set it off from other text. However, the
capitalization of letters is not significant in Pascal (except
within constants, as we will see).

Symbols

Symbols are the smallest meaningful elements in Pascal,
corresponding to the words, spaces, and punctuation in an English
sentence. Everything else in the language is built up out of
symbols.

Reserved Words

These symbols are words such as FOR, WHILE, AND, DO, and BEGIN.
They make up the essential vocabulary of Pascal and have fixed
meanings. A complete table of reserved words is given in
Appendix J.

Identifiers

These symbols are names for things such as variables, data types,
and procedures. Most identifiers are made up by the programmer

and given meanings in declarations; other identifiers are names
of variables, data types, procedures, etc., that are built into
the language and don't need to be declared. The syntax for an
identifier is

identifier
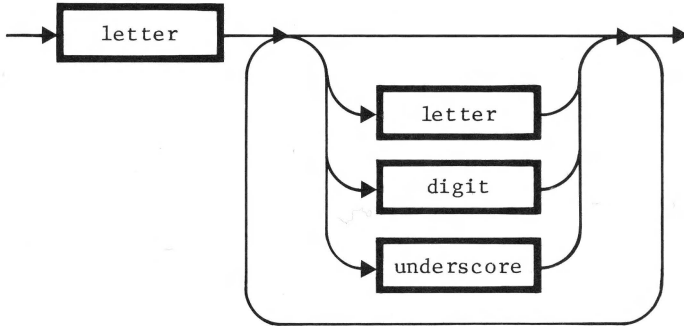
Thus an identifier must begin with a letter, and this letter may
be followed by any number of letters, digits, or underscores.
The Compiler ignores the case of letters; 'A' and 'a' are
equivalent. Underscore characters in an identifier are ignored;
their only purpose is to make an identifier more legible. Also,
only the first eight characters (not counting underscores) are
significant. For example, the following six identifiers are
equivalent and interchangeable:

    MYNUMBER   mynumber
    MY_NUMBER   My_Number
    MY_NUMBER_VALUE   MY_NUMBER_SYMBOL

There is an important restriction on identifiers: an identifier
must not be the same as any reserved word.

Numeric Constants

Numeric constants are signed decimal integer or floating-point
values that are written into the program to be treated as data.
In Pascal there are two kinds of numeric data called integer and
real. Integer constants must fall in the range -32767 to 32767.
The following are valid integer constants:

    -1   2   865   16383 -2000 0 1949

A real constant contains a decimal point (period) and may contain
an "exponent part." The exponent part consists of the letter "E"
and a number, and indicates multiplication by a power of ten.
The following are valid real constants, and all represent the
same numerical value:

    3.14159 Ø.314159E1 3141.59E-3

Character Constants

These are single characters written into the program to be
treated as data. The apostrophe or "single quote" is used to set
off character constants. The following are examples:

    'a' 'A' 'Ø' '+' ''''

The first two constants, 'a' and 'A', are not equivalent.
Capital letters are distinguished from lower case in constants.
The last example shows how to represent a single quote as a
character constant. By the way, don't confuse two single quotes
with the double-quote character ("), which has no meaning in
Pascal.

String Constants

These are character sequences written into the program to be
treated as data. Like character constants, they are set off by
apostrophes. Examples:

    'Smith' '$4Ø8.23' 'Type your name: 'DON''T WORRY' ''

Capital letters are distinguished from lower case. The
next-to-last example shows how to use an apostrophe as a
character within a string constant. The last example shows how
to represent a string consisting of no characters.

Delimiters

Delimiters have the effect of separating other symbols from each
other, which is why they are called delimiters. When two symbols
are not separated by a delimiter, they must be separated by a
space or a line break.

These special characters (and a few two-character combinations)
also have various special meanings such as arithmetic operations,
array indexing, setting off comments. etc. The one-character

delimiters are

,   .   ;   :   '   (   )   [   ]   +   -   /   *   =   <   >   ^   {   }

and the two-character delimiters are

:=   ..   (*   *)   <=   >=   <>

Declarations

You can think of a Pascal program as being made up of two kinds
of "sentences"--statements, which generally cause some kind of
action to occur when the program is executed, and declarations.
Use declarations to announce the nature of an identifier.

Variable Declarations

All variables must be declared (except dynamic variables, which
are discussed later). The effect of a variable declaration is to
create an identifier, associate it with a data type, and allocate
memory space for it. Variable declarations are introduced by the
reserved word VAR. Example:

```
VAR NEWVALUE:INTEGER;
    RESULT:REAL;
```

These two declarations create an identifier NEWVALUE which is the
name of an integer variable, and an identifer RESULT which is the
name of a real variable.

Constant Declarations

Identifiers may also be declared for numeric and character
constants. A constant identifier can then be used in the program
instead of the constant itself. Constant declarations are
introduced by the reserved word CONST. Example:

```
CONST PI=3.14159;
      ROWSIZE=64;
      COLUMNSIZE=2048;
```

These three declarations create the identifiers PI, ROWSIZE, and
COLUMNSIZE, with the values indicated. In a program that uses
the value of pi in many different statements, it is more

convenient to declare PI as shown and use it in the statements
instead of having to write 3.14159 repeatedly.   Another advantage
of declared constants is that if a program is first developed
with ROWSIZE and COLUMNSIZE as shown above, and later you want to
change these values, you need only change the declarations rather
than searching through all the statements for the values 64 and
2Ø48.

Type Declarations

Pascal allows you to declare your own data types, as described in
the next section.   Type declarations are introduced by the
reserved word TYPE.

Data Types

Pascal has a great variety of data types.   They are described
very briefly here, and several chapters are devoted to describing
them in detail.   The data types are broken into two broad
categories called "simple" and "structured" types; a simple data
type represents just one value, while a structured type
represents a collection of values.

Most of the simple types are "scalar" types.   This important term
is defined in Chapter 3.

The Real Type

Real values are signed, 32-bit floating-point numbers in the
proposed IEEE format with a precision of about seven decimal
places (depending on the actual value).   The range of real values
is from plus or minus 1.4Ø1298464E-45 to plus or minus
3.4Ø2823466E38; also, Ø.Ø is a real value.   (See Appendix E for
complete details.)

The Integer Type

Integer values are signed whole numbers in the range -32768 to
32767.   There are also "long integers" which are BCD-coded and
can represent values up to 36 decimal digits.

The Char Type

A value of type char is a character: that is, any member of the
extended 8-bit ASCII character set used on the Apple III.

The Boolean Type

A boolean value is either TRUE or FALSE. Such values result from
various kinds of expressions (such as comparisons), can be used
with the logical operators NOT, AND, and OR to result in new
boolean values, and are used in some of the control statements of
Pascal. Note that TRUE and FALSE are not numerical values.

User-Defined Scalar Types

In Pascal, a scalar data type is one that has a fixed number of
possible values which form a fixed sequence. The integer, char,
and boolean types are scalar types (the sequence for char values
is the sequence of the ASCII table, and the sequence for booleans
is FALSE, TRUE).

You can also define your own scalar types. For example, the
declaration

        VAR DAY:(SUN, MON, TUES, WED, THURS, FRI, SAT);

creates a variable called DAY. The possible values of DAY are
represented by the identifiers SUN, MON, etc. The actual values
are internal bit patterns, of course, but you don't need to know
what they are because you have the identifiers for them. The
values have the ordering indicated in the declaration.

Subrange Types

Subrange types are variations on the scalar types described
above. For example, the declaration

        VAR DAYNUM:1..7;

creates an integer subrange variable whose values cannot be less
than 1 or greater than 7. If the program tries to give DAYNUM a
value outside this range, the interpreter will halt the program
with an error message. Subrange types can also be based on the
char type and on user-defined scalar types, as shown in the
following declarations:

```
VAR CAPITALS:'A'..'Z';
    DAY:(SUN, MON, TUES, WED, THURS, FRI, SAT);
    WEEKDAY:MON..FRI;
```

Subrange types are particularly useful for array indices.

The String Type

The value of a string variable is a sequence of char values.
Besides the value, a string variable also has a length attribute
which is automatically maintained by the system when the value of
the string changes during program execution. The declaration

```
VAR LASTNAME: STRING;
```

creates a variable LASTNAME of type string. The maximum length
for a string is 255 characters. Apple III Pascal has a powerful
set of built-in procedures and functions for manipulating string
values.

Array Types

An array is a collection of values, all of the same type. The
values are called array elements. A particular element of an
array can be referenced by writing the name of the array followed
by one or more indices enclosed in square brackets []. The array
is said to have one dimension for each of its indices.

Pascal arrays are very flexible. They can have any number of
dimensions (subject to space restrictions). The elements of an
array can be of any type except file types. The indices of an
array can be of any subrange type (including integer subranges),
or any scalar type except integer. The declaration

```
VAR NUMS: ARRAY[0..511] OF REAL;
```

creates a one-dimensional array named NUMS, with 512 elements.
Each element is a real value. The first element in the array is
NUMS[0] and the last is NUMS[511]. The declaration

```
VAR MATRIX: ARRAY[1..255, 1..64] OF BOOLEAN;
```

creates a two-dimensional 255x64 array of boolean values. Note
that this time, the indices start at 1 instead of 0. Finally,
the declaration

```
VAR CHARCODE: ARRAY[CHAR] OF 0..255;
```

creates a one-dimensional array which contains values in the
range 0 to 255. The array is indexed by characters. For
example, CHARCODE['A'] refers to the 66th element of the array,
since the character A is the 66th character in the ASCII set.

Record Types

Like an array, a record variable is a collection of values; but
the values belonging to a record need not all be of the same
type. Each element of a record variable has its own identifier.
The declaration

```
VAR PERSON: RECORD
              FIRSTNAME: STRING;
              INITIAL: CHAR;
              LASTNAME: STRING;
              AGE: INTEGER
            END;
```

creates a record variable PERSON which contains four values; two
of the values are strings, one is a char, and one is an integer.
The elements of PERSON are referred to in the program as
PERSON.FIRSTNAME, PERSON.INITIAL, PERSON.LASTNAME, and
PERSON.AGE.

Set Types

The declaration

```
VAR SPECIALCHARS: SET OF CHAR;
```

creates a set variable SPECIALCHARS which is a set of
characters. We say that the type char is the base type of the
set SPECIALCHARS. During program execution, the actual value of
SPECIALCHARS is a bit pattern which reflects the presence or
absence of each character in the ASCII character set; to make use
of this value, Pascal provides a notation for writing a set value
as in the following assignment statement:

```
SPECIALCHARS := ['.', ':', ';']
```

This statement assigns to SPECIALCHARS the set of characters
consisting of the period, colon, and semicolon characters.
Pascal also provides operations for adding a new member to a set,

testing to see if a particular value is a member of the set,
forming the union, intersection, or difference of two sets, and
comparing sets. The base type of a set can be any subrange type
(including integer subranges), or any scalar type except
integer. The base type cannot have more than 512 possible
values; this means that a set cannot have more than 512 members.

Dynamic Variables and Pointers

Pascal provides a mechanism for using variables that are created
during program execution, using unallocated memory space; these
are called dynamic variables. A dynamic variable can be of any
data type (except a file type, which is discussed in the next
section). A dynamic variable is not declared; instead, a pointer
variable is declared. During program execution, the pointer is
used to create one or more dynamic variables. For example,
consider the following declarations:

```
TYPE PREC=RECORD
            FIRSTNAME: STRING;
            INITIAL: CHAR;
            LASTNAME: STRING;
            AGE: INTEGER
          END;

VAR PPTR: ^PREC;
```

The type PREC is equivalent to the type of the PERSON variable in
the example for record types (see above). The pointer variable
PPTR is defined as a pointer to a dynamic variable of type PREC.
The program can then contain the statement

```
NEW(PPTR)
```

which creates a new dynamic variable of type PREC. This variable
can be referred to in the program as PPTR^, and its elements as
PPTR^.FIRSTNAME, PPTR^.INITIAL, etc.

File Types

In Pascal, files are considered to be program variables. A file
is considered to be a sequence of an indefinite number of data
elements, all of the same type. The elements can be of any type
except a file type. The declaration

```
VAR OUT: FILE OF INTEGER;
```

creates a file variable OUT whose elements are of type integer. Physically, a file in the Apple III system is either a diskette file or some device or device driver attached to the system; so Apple III Pascal has mechanisms for associating a file variable with an external file. A powerful set of built-in procedures is provided for doing file I/O.

Statements

Pascal statements do the work of a Pascal program. There are ten distinct kinds of statement in Pascal; they are described here very briefly. A distinctive feature of Pascal is that some kinds of statements can contain other statements.

Pascal uses the semicolon character in two ways. In the preceding sections we have seen semicolons used to terminate declarations. In the body of a program or procedure, Pascal uses semicolons to separate statements. When a semicolon appears at the end of a statement, it is not part of the statement: it stands alone and separates the statement from the next statement. If the next thing in the text is not a statement (for example, the word END), no semicolon is required. The examples in this book use semicolons only where they are actually required.

The Assignment Statement

An assignment statement gives a value to a variable. An example is

    MEANVAL := (LEFTVAL + RIGHTVAL) / 2

where MEANVAL, LEFTVAL, and RIGHTVAL are variables of type real. The effect of this assignment statement is to calculate the sum of the current values of LEFTVAL and RIGHTVAL, divide by 2, and make the resulting value the new value of MEANVAL.

The Procedure Call Statement

Procedures are discussed further on; they can be thought of as subprograms that are embedded in the main program. The statement

    WRITE('ABCD')

is a procedure call statement that activates the WRITE
procedure. WRITE is a built-in procedure; that is, it is
automatically available to all Apple III Pascal programs. In the
example, the procedure call statement passes the string ABCD to
the WRITE procedure as a parameter; WRITE will display the string
on the screen.

The Compound Statement

In some ways this is the most important kind of statement in
Pascal. It consists of the reserved word BEGIN, followed by any
number of Pascal statements, followed by the reserved word END.
This allows you to put any number of statements in a place where
only one statement is allowed, since the compound statement is
just one statement. You will see many examples of this
practice. Also, the body of every program and of every procedure
and function definition is a single compound statement.

The IF Statement

The IF statement provides conditional execution of a statement
which is contained in the IF statement. The condition is the
value of an expression, which has a boolean value (TRUE or
FALSE). An example:

        IF A < B THEN C := D

In this example, the assignment statement C := D will be executed
only if the expression A < B has the value TRUE--that is, if the
value of A is less than the value of B. Another example:

        IF X = Y THEN BEGIN
                      MATCHCNT := MATCHCNT + 1;
                      WRITELN('X is equal to Y')
                    END

This shows the use of a compound statement, which will be
executed only if X and Y have equal values. Inside the compound
statement are two statements—an assignment statement and a
procedure call. Note the use of a semicolon to separate the two
statements. Also, note that just as the compound statement is a
single statement containing two other statements, the entire IF
statement is a single statement containing the compound
statement.

An IF statement can also have an ELSE part, which is executed only if the condition is FALSE; for example,

```
IF M = Ø THEN ZEROCNT := ZEROCNT + 1
         ELSE NONZEROCNT := NONZEROCNT + 1
```

Here either ZEROCNT or NONZEROCNT will be incremented, depending on whether or not M is equal to Ø.

The CASE Statement

The CASE statement lets a program select and execute one statement from a set of statements. The CASE statement contains an expression, and the statements inside the CASE statement are labeled with possible values of the expression; thus the value of the expression selects one statement to be executed. An example:

```
CASE COMMCH OF
   'A':   APPEND;
   'R':   REORDER;
   'F':   FORGET;
   'X':   BEGIN
             WRITELN('Exiting from program');
             EXIT(PROGRAM)
          END
END
```

where the controlling expression is COMMCH (a char variable) and APPEND, REORDER, FORGET, and EXIT are procedures. If COMMCH is 'A', the first statement is executed, calling the APPEND procedure. If COMMCH is 'R', the second statement is executed to call the REORDER procedure, and so forth. Note the use of a compound statement to do two things in the 'X' case. If COMMCH does not match any of the labels in the CASE statement, none of the statements within the CASE statement are executed; the program proceeds to the next statement after the CASE statement.

The REPEAT and WHILE Statements

These statements, like the IF statement, contain an expression whose value is boolean (TRUE or FALSE). A REPEAT statement contains one or more statements which are executed once, and then repeated if the expression's value is FALSE. Repetition continues until the expression's value is TRUE. The following example prompts the user to type the word "cat" and repeats until

the user does so:

```
REPEAT
  WRITE('Type the word "cat"  ');
  READLN(INSTRING)
UNTIL INSTRING = 'cat'
```

In this example, the built-in procedure READLN reads whatever the user types (terminated with the RETURN key) and puts the characters in INSTRING.

The WHILE statement executes a single (possibly compound) statement repeatedly, as long as the value of the controlling expression is TRUE. This value is tested before anything is executed; thus in some cases the statement within the WHILE statement may not be executed at all. The following example is a variation on the previous one; note that the symbol <> means "not equal to":

```
WRITE('Type the word "cat"  ');
READLN(INSTRING);
WHILE INSTRING <> 'cat' DO
  BEGIN
    WRITELN('You typed ', INSTRING);
    WRITELN('Try again:');
    WRITE('Type the word "cat"  ');
    READLN(INSTRING)
  END
```

The compound statement in the WHILE statement is executed only if the user fails to type "cat" on the first try; but in this event the compound statement is executed until the word "cat" is typed.

The FOR Statement

The FOR statement controls repetition of a single (possibly compound) statement by counting. For example, the following statement will list the integers from 1 to 1∅ on the screen:

```
FOR NUMBER := 1 TO 1∅ DO WRITELN(NUMBER)
```

where NUMBER is an integer variable which is used as the <u>control variable</u> of the FOR statement. First the value 1 is assigned to it. Then the WRITELN statement is executed (using the current value of NUMBER). Then the value of NUMBER is changed to the

next integer in sequence, and the WRITELN is executed again.   The
process is repeated until the value of NUMBER exceeds the value
of the <u>limit expression</u>, which is 1∅ in this case.   Thus the last
number written on the screen is 1∅.

The FOR statement can also count in reverse, by using the word
DOWNTO instead of TO.   The following will list the letters in
reverse order from Z to A:

    FOR CHARACTER := 'Z' DOWNTO 'A' DO WRITELN(CHARACTER)

Notice that in this example character values are used instead of
integer values.   In fact, any scalar type can be used to control
a FOR statement.

The WITH Statement

The WITH statement is a convenience for referring to the fields
within a record variable without having to write the identifier
of the record repeatedly.

The GOTO Statement

The GOTO statement causes an unconditional "jump" to a specific
statement.   There are some important restrictions on the use of
GOTO in Apple III Pascal; see Chapter 5 for details.

Expressions

Pascal expressions are "algebraic" in form.   An expression can
contain just a single value, such as a variable identifier or a
constant, or it can be a complex combination of various operands
and operators.

Arithmetic Operators

For arithmetic operations with real and integer operands, the
operators are

```
+           addition
-           subtraction or negation
*           multiplication
/           division with real result
DIV         division of integers with integer result
            rounded toward zero
MOD         remainder of integer division
```

In Apple III Pascal, all arithmetic operations on real values conform to the proposed IEEE floating-point standard. See Appendix E for complete details.

Comparison Operators

Comparisons between scalar or real operands, yielding boolean results, are performed by the operators

```
>           greater than
>=          greater than or equal to
=           equal to
<=          less than or equal to
<>          not equal to
```

Logical Operators

Logical operations with boolean operands and boolean results are performed by the operators

```
NOT         boolean negation
AND         boolean conjunction
OR          boolean disjunction
```

Set Operations

Set operations with set results are performed by the operators

```
+           union
-           difference
*           intersection
```

Set comparisons, with boolean results, are performed by the operators

```
=          equal to
<>         not equal to
>=         includes or is equal to
<=         is included by or equal to
IN         is a member of
```

Notice that some of the operator symbols are used in more than one way: for example, the + symbol can be the numeric addition operator or the set union operator. In such a case, the meaning of the operator symbol depends on the operands.

Procedures

A procedure can be thought of as a "subprogram," which is defined (that is, written) inside the main program (before the main program's body of executable statements) and can then be executed by the main program, using a procedure call statement.

An example of a procedure definition:

```
PROCEDURE TWOLINES;
  BEGIN
    WRITELN;
    WRITELN;
  END;
```

This procedure does nothing but call the WRITELN procedure twice; this displays two blank lines on the screen. A more useful procedure would be

```
PROCEDURE NLINES (NUMBER: INTEGER);
  VAR COUNTER: INTEGER;
  BEGIN
    FOR COUNTER:=1 TO NUMBER DO WRITELN
  END;
```

This procedure has a parameter: an integer value which is called NUMBER within the procedure definition. The value of NUMBER is determined when NLINES is called by a statement such as

```
NLINES(5)
```

which gives the value 5 for the parameter. NLINES uses the value of NUMBER as the limit value in a FOR statement; the procedure

call statement NLINES(5) would cause NLINE to put five blank
lines on the screen.

Notice that in order to use the FOR statement, NLINES needs an
integer variable to control it. Therefore NLINES contains a
declaration of the integer variable COUNTER. This variable
belongs to NLINES and is unknown to the rest of the program. In
fact, the main program could have another variable with the same
name, COUNTER, and this would not cause any problems.

Every procedure has the following general outline:

> procedure heading,
>   consisting of the word PROCEDURE, the procedure's name,
>   and a list of any parameters;
>
> declarations of any user-defined data types,
>   constants, variables, etc.
>
> definitions of any procedures and functions
>
> the word BEGIN
>
>   any number of statements, separated by semicolons
>
> the word END, followed by a semicolon

Compare this to the structure of a program shown in the previous
chapter. The underlines indicate the only differences between a
procedure definition and a program. Note that procedure
definitions can be written inside other procedure definitions;
when this is done, the inner procedure belongs to the outer one,
and is unknown to the rest of the program.

For more about procedures, see Chapter 6.

Functions

A function is a procedure that returns a value. While a
procedure call is a statement, a function call is an operand in
an expression. The function calculates a value, and this value
becomes the value of the operand in the expression. The
following function calculates the cube of a real value:

```
FUNCTION CUBE (V: REAL): REAL;
  BEGIN
    CUBE := V * V * V
  END;
```

Note that the function heading includes the type of the value calculated by the function. Within the function, the function name appears on the left side of an assignment statement; this establishes the value that the function returns. The CUBE function could be used in an expression as follows:

```
CUBE(5.7*Y) + 2.3
```

In evaluating this expression, the value of Y is first multiplied by 5.7. The result is the parameter value to be passed to CUBE. The CUBE function is then executed, and it calculates the cube of the parameter value. This value then becomes the first operand in the expression; 2.3 is added to it to get the value of the expression.

For more about functions, see Chapter 6.

Built-In Procedures and Functions

Pascal has a large set of procedures and functions built into it. These procedures and functions do not have to be defined; they do not reside in a library; they are part of the language itself and are automatically available to all programs. They serve such purposes as control of input and output, various mathematical functions, manipulation of strings, use of dynamic variables, and various special purposes. Further information on the built-in procedures and functions is in various chapters, along with the topics that the procedures and functions relate to.

Pascal Program Structure

Chapter 1 gives an outline of the structure of a Pascal program. We can now show the structure in a slightly more sophisticated way. First we define something called a block.

Every block has the following outline:

    optional label declarations—the word LABEL followed by
     declaration of labels

    optional constant declarations—the word CONST followed by
     declarations of identifiers for constants

    optional type declarations—the word TYPE followed by
     declarations of user-defined data types

    optional variable declarations—the word VAR followed by
     declarations of variables

    optional definitions of procedures and functions

    one compound statement

The LABEL declaration has been mentioned briefly; it is only
needed in a block that uses GOTO statements.

The idea of a block is important in Pascal. For one thing, we
can now say that the outline of every program is

    program heading

    optional USES declaration

    block

    period

(The USES declaration is explained below.) Similarly, the
outline of every procedure definition is

    procedure heading

    block

    semicolon

and the outline of every function definition is

function heading

block

semicolon

The Scope of Identifiers

The scope of an identifier is simply the part of the program in
which it is known. Here are the rules about the scope of
identifiers:

- An identifier that is declared in a procedure or
  function is not known outside of that procedure or
  function. This includes the identifiers of parameters
  in the procedure or function heading.

- A procedure or function can re-declare an identifier
  which was already declared outside the procedure or
  function. In this case there are two different things
  that happen to have the same identifier, but there is
  no problem: the "outer" identifier is unknown inside
  the procedure or function, and vice versa.

- The identifier of a procedure or function is considered
  to be declared both inside and outside the procedure or
  function. That is, the identifier is known both inside
  and outside and you cannot re-declare it.

To see how this works, consider the following fragment of a
program:

```
PROGRAM SAMPLE;

CONST MSG = 'Limit exceeded.';

VAR LIMIT:REAL;

PROCEDURE TSTLIM (X: INTEGER);
  CONST LIMIT=3;
  BEGIN
    IF X > LIMIT THEN WRITELN(MSG)
  END;

BEGIN
  .
  .
  .
```

The string constant MSG is known throughout the program, including the procedure TSTLIM. We say that MSG is <u>local</u> to the program and <u>global</u> to TSTLIM.

The integer variable X, declared in the procedure heading of TSTLIM, is known only inside TSTLIM. We say that X is <u>local</u> to TSTLIM and <u>unknown</u> to the program.

The real variable LIMIT is known throughout the program, <u>except</u> inside TSTLIM, because the integer constant LIMIT is declared within TSTLIM. The integer constant LIMIT is local to TSTLIM and unknown to the program—while the real variable LIMIT is local to the program and unknown to TSTLIM.

The example shows a procedure within a program, but exactly the same rules apply when a procedure or function is nested within another procedure or function.

This scoping of identifiers becomes a real advantage when a program is large or complex. It means that you (or some other programmer) can develop a procedure without worrying about whether the procedure's variables will conflict with the main program's variables.

Library Units

A library unit is a package of "public" procedures, functions, variables, etc. that has been compiled separately and placed in a library file. A file named SYSTEM.LIBRARY is an integral part of

the system and contains a number of standard packages. These
packages provide such things as transcendental functions,
graphics functions, and access to special Apple III machine
features.

A program that uses a library unit has access to all the public
features of that unit, just as if they were declared at the
beginning of the program. To use a unit, the program merely
gives the name of the unit in a USES declaration immediately
after the program heading. For example, two of the standard
units are called TRANSCEND and APPLESTUFF. To use both of them,
a program would have the declaration

     USES TRANSCEND, APPLESTUFF;

immediately after the program heading.

Sample Program

At the end of Chapter 1, a sample program was given without
explanation. Now we can explain how the program works.

```
PROGRAM FIRSTEXAMPLE;                   { program heading }

VAR I:INTEGER;                          { declaration of
                                          a variable }

PROCEDURE DISPLAY (J:INTEGER);     { procedure definition }
  BEGIN
    WRITELN;
    WRITELN('The number is ', J)
  END;                                  { end procedure
                                          definition }

BEGIN                              { begin program body }
  FOR I:=Ø TO 1Ø DO DISPLAY(I)           { statement }
END.                                 { end of program }
```

One procedure, DISPLAY, is declared. DISPLAY accepts one
parameter, an integer value which is known within the procedure
as J.

Notice that the program body contains just one statement: a FOR
statement that repeats for values from Ø through 1Ø. The FOR

statement is controlled by the integer variable I, declared in
the VAR declaration at the beginning of the program.

Each time the FOR statement executes, it executes a procedure
call to the DISPLAY procedure. The procedure call passes the
current value of I to DISPLAY. DISPLAY assigns this value to its
own variable J. DISPLAY then executes two statements:

 - First it calls the built-in procedure WRITELN with no
   parameters. This causes a blank line to be displayed
   on the screen.

 - Then it calls WRITELN again with two parameters
   (separated by the comma). The first parameter is the
   string constant 'The number is ', and the second is the
   integer value J. WRITELN displays these values on one
   line on the screen.

Each time DISPLAY is called, the value of I is passed to it as
the value for J. Since I is the control variable of the FOR
statement, it takes on the values 0 through 10 in sequence.
Therefore, the output from the program is

      The number is 0

      The number is 1

      The number is 2

      The number is 3

      The number is 4

      The number is 5

      The number is 6

      The number is 7

      The number is 8

      The number is 9

      The number is 10

3

Simple Data Types

Introduction

Every piece of data used or created by a Pascal program has an attribute called its type. The type tells the program and the system how to interpret the piece of data. For example, one of the fundamental types is the integer type; another distinct type is the real type. An integer value is a signed whole number, while a real value is a signed floating-point number. You can think of a data type as a definition of the set of possible values that the data can have.

If the piece of data is the value of a named constant or variable, its type is assigned when the constant or variable is declared. Otherwise, the type of the data depends on how the program inputs or creates the data.

Pascal offers an especially wide range of built-in types, each with its own identifier. Pascal also allows you to define your own types within a program. A user-defined type can be a composite of built-in types, for example, or it can be a subrange of a built-in type. In this chapter, we cover the simple data types—i.e. the types that a single value can have. These are

        The type REAL
        The scalar types:
                INTEGER and LONG INTEGER
                CHAR
                BOOLEAN
                User-defined scalar types
                Subrange types

Other chapters will cover the structured data types, where a typed piece of data can be a collection of various single values.

        Jensen and Wirth, in their definition of Pascal, define the type REAL to be a scalar type. Most books on Pascal follow Jensen and Wirth on this point. But REAL values are not handled in the same way as values of other scalar types, and are an exception to most of the rules about scalars; therefore, this book considers reals to be a separate category.

Declarations

As described in the previous chapter, the structure of a Pascal program includes sections for several different kinds of declarations. Here is a sample program that uses two kinds of declarations:

```
PROGRAM XYZ;

CONST MAXA = 24;            {An integer constant}
      MAXB = 31;            {Another integer constant}
      COEFFICIENT = 17.3;   {A real constant}

VAR A : INTEGER;           {An integer variable}
    B : INTEGER;           {Another integer variable}
    X : REAL;              {A real variable}

BEGIN
  FOR A := Ø TO MAXA DO
    FOR B := Ø TO MAXB DO
      BEGIN
        X := A + B*COEFFICIENT;
        WRITELN(X)
      END
END.
```

Note that different kinds of declarations must be placed in a specific order in the program. Variables are always declared last, and constants are always declared before variables. There are also several other kinds of declarations; further on in this chapter we will see type declarations.

Declaring Constants

Generally, constants do not have to be declared; however, it is often a convenience to do so. When you declare a constant, you are creating an identifier and associating it with a specific, unchanging value. All constant declarations are grouped together and introduced by the reserved word CONST.

The syntax diagram for constant declarations is

constant declarations
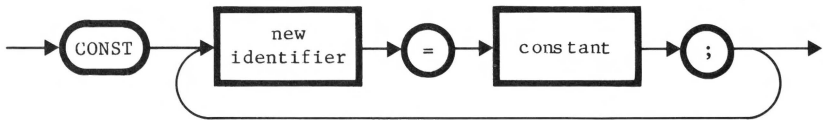
Note the use of the "=" symbol, and note that each declaration
ends with a semicolon. In this diagram, "constant" can be any of
the following:

- A signed or unsigned whole number representing a value
  of type INTEGER. (See section further on for LONG
  INTEGER numbers).

- A signed or unsigned floating-point number representing
  a value of type REAL.

- A character or string of characters enclosed in
  apostrophes, representing a value of type CHAR or type
  STRING. A single character string constant is
  identical to a constant of type CHAR. Strings are
  explained in Chapter 7.

- The identifier of a previously declared constant (to
  create a new constant with the same value and a
  different identifier).

The following example declares two constants:

        CONST PI = 3.14159;
              MAXITERATIONS = 1Ø;

The first constant, PI, has the value 3.14159, and the second,
MAXITERATIONS, has the value 1Ø. PI is a constant of type real
and MAXITERATIONS is a constant of type integer.

You can use the constant PI wherever a real value is allowed; for
example, in the expression

        2 * PI * RADIUS

where RADIUS is a variable. In this expression, 2 is an example
of an integer constant that has not been declared. Incidentally,
the integer value 2 is automatically converted to the real value

2.∅ before the multiplication takes place (for details, see Chapter 4).

Declaring a constant such as PI is a convenience, not a necessity; you could write

    2 * 3.14159 * RADIUS

and it would mean exactly the same thing.

Declaring Variables

All variables <u>must</u> be declared. When you declare a variable, you are creating an identifier and associating it with a specific data type; when the program is executed, the variable can take on any of a set of values depending on the type. All variable declarations are grouped together and introduced by the reserved word VAR.

The following example declares two variables:

    VAR RATIO: REAL;
        ITERATION: INTEGER;

Again, each declaration ends with a semicolon. The first variable, RATIO, is of type real, and the second, ITERATION, is of type integer. When two or more variables of the same type are declared, you can combine the declarations:

    VAR I, J, K: INTEGER;
        X, Y, Z: REAL;

This declares three integer variables, I, J, and K, and three real variables, X, Y, and Z.

In the remainder of this chapter, you will see many examples of variable declarations, using many different data types. However, all of them follow the general form of the examples just given. The syntax diagram for variable declarations is

variable declarations
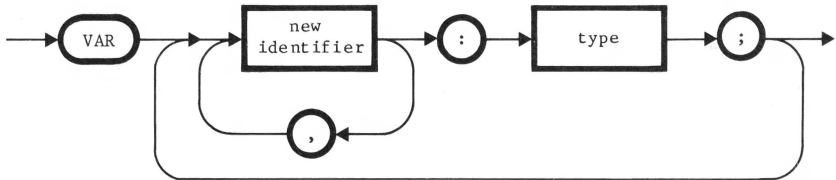
Note the use of the ":" symbol, and note that each declaration ends with a semicolon. The word "type" in the diagram stands for any of a wide range of possibilities. In this chapter we are concerned with certain predefined types, which are represented by identifiers such as INTEGER and REAL; we will also introduce two of the ways in which you can define new data types.

The Real Type

A value of type real is a signed floating-point number. It is stored as a 32-bit number following the IEEE format. Real values can be combined with each other by means of the arithmetic operators +, -, *, and / to yield real results. A real value can also be combined arithmetically with an integer value; when this happens, the integer value is first automatically converted to a real value, and the result is real.

An integer value is also automatically converted to a real value when it is assigned to a real variable. A real value can be converted to an integer value in either of two ways: the TRUNC function and the ROUND function. These built-in functions are described in the last section of this chapter.

A real value can be compared arithmetically with another real value or with an integer value by means of the $<$, $<=$, $=$, $>=$, $>$, and $<>$ operators, to yield boolean results.

The range of real values is from plus or minus 1.4Ø1298464E-45 to 3.4Ø2823466E38; Ø.Ø is also a real value. Each real value is represented in 32 bits (two 16-bit words, or four 8-bit bytes). This gives a precision of about 7 significant digits (depending on the actual value).

Warning: do not confuse the Pascal type real with the
mathematical idea of a real number. The Pascal type is a
floating-point bit pattern or code which can be used to
represent a number in the computer. These codes and the
operations on them do not always correspond exactly to
their mathematical counterparts.

For example, if the exact result of an operation on reals
can't be represented in the 32-bit format, it is
automatically rounded to fit. Subsequent calculation with
this rounded value may make later results approximate as
well. It is a wise precaution to analyze your program
carefully, anticipating errors from this cause. See
Appendix E for further information.

Declaring Real Variables

As shown in the examples above, real variables are declared by
using the word REAL to the right of the colon in the
declaration.

Declaring Real Constants

A constant of type real can be declared in a constant declaration
with a literal floating-point number to the right of the equal
sign (read on for examples). The syntax for a floating-point
number is

floating-point number
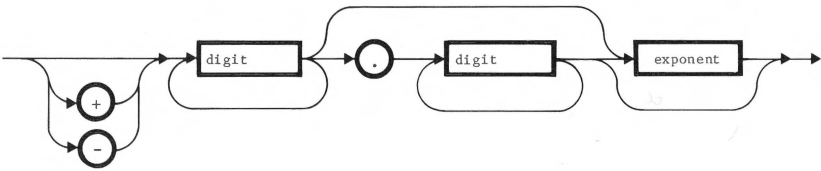
and the syntax for an "exponent" is

exponent
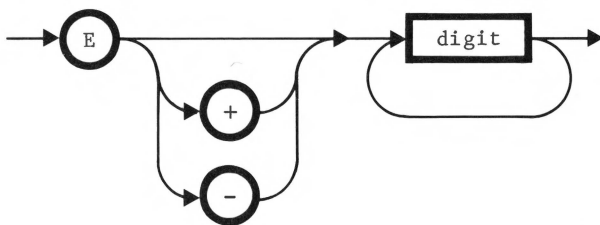
Examples are

> 3.14159   2.0   -3.5   12.6E3   7.8236E-12   -83769E-3

The "E" notation indicates a power of ten; for example, 12.6E3
and 12600.0 mean the same thing. Note that the last example has
no decimal point; this shows that if a numeric constant has
either a decimal point or an "E" in it, it is considered to be a
real constant. Finally, note that if there is a decimal point,
it must have at least one numeric digit on either side of it.

Scalar Types

A scalar data type has a distinct set of possible values, which
are considered to be ordered in a specific way: they can be put
in one-to-one correspondence with some sequence of possible
values of type integer. There are three built-in scalar types,
namely integer, char, and boolean; these are described below.
Also, you can create your own scalar types.

In Apple III Pascal, the type real is not a scalar type by this
definition; the reason is that there are not enough integer
values to match up with all the possible real values.

The Integer Type

Integer values are whole numbers in the range from -32768 through
32767. Integer values can be combined with each other by means
of the arithmetic operators +, -, *, DIV, and MOD to yield
integer results, or by means of the / operator to yield a real

result. An integer value can also be combined arithmetically with a real value; when this happens, the integer value is first automatically converted to a real value, and the result is real.

An integer value can be compared arithmetically with another integer value or with a real value by means of the <, <=, =, >=, >, and <> operators, to yield boolean results.

An integer value is automatically converted to a real value when it is assigned to a real variable.

Declaring Integer Variables

As shown in the examples above, integer variables are declared by using the reserved word INTEGER to the right of the colon in the declaration.

Declaring Integer Constants

A constant of type integer can be an identifier previously declared in a constant declaration or a non-floating-point number. The syntax for a non-floating-point number is

non-floating-point number
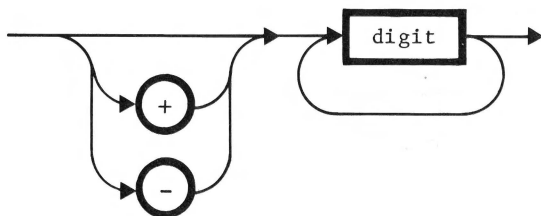
Examples are

    1 Ø -3 23583 -4532

Note that a non-floating-point number is one that contains neither a decimal point nor the letter "E". Note also that -32768 cannot appear as an integer constant in a program.

MAXINT

MAXINT is the identifier of a built-in constant whose value is the largest possible integer, 32767.

The Long Integer Type

This is a special-purpose type. A long integer variable contains a signed whole number represented internally as a sequence of up to 36 binary-coded decimal (BCD) digits. The maximum number of digits is specified by a length attribute in the declaration of the variable. For example, the declaration

    VAR CENTS: INTEGER[9];

creates a long integer variable named CENTS which can represent a value of up to 9 decimal digits. You can specify a length attribute up to and including 36. (The actual limit, at run time, may be greater than the number you specify.)

There are also long integer constants; any constant whose value is a non-floating-point number greater than 32767 (the largest integer) or less than -32768 (the least integer) is a long integer constant.

An integer value can be assigned to a long integer variable; it is automatically converted to a long integer value. A long integer value cannot be directly assigned to an integer variable; however, if it is not greater than 32767 or less than -32768 it can be converted to an integer value by means of the TRUNC function described further on in this chapter.

Long integers cannot be used as freely as integers. Long integers can be combined with each other by means of the +, -, *, and DIV operators (but not the / and MOD operators), to give long integer results. With the same operators, a long integer value can be combined with an integer value; the integer value is automatically converted to a long integer and the result is of type long integer.

In long integer arithmetic, overflow occurs if any intermediate or final result would exceed 36 digits. This causes a run-time error halt.

Note that long integer is not a scalar type, for the same reason that real is not a scalar type—there are more possible long integer values than there are integer values. Long integers are mentioned here simply because they are conceptually related to integers.

The standard system library file, SYSTEM.LIBRARY, contains a library unit called LONGINTIO. Long integer operations cannot be executed unless this unit is available to the program at start of execution.

The Char Type

A char value is any character from the 8-bit ASCII character set used on the Apple III. Thus the char values correspond to the integers from 0 to 255; the integer associated with each char value is called its ASCII code. The first 128 char values (ASCII 0 through ASCII 127) have standard interpretations as printing characters and control characters, as shown in the ASCII code table in Appendix J. The remaining char values can also be used as explained below in the section on the CHR function.

A char variable is declared by using the reserved word CHAR in the declaration. For example, the declaration

        VAR CURRENT_CH, LAST_CH: CHAR;

declares two variables (CURRENT_CH and LAST_CH) of type char.

A character constant is formed by placing the character between apostrophes (single quotes), as in the following examples:

        'a'   '5'   ''''   ','   ' '

Note that the value of the third constant is the apostrophe character; this is a special notation. The value of the last constant is the space character.

A character constant can be declared, as in the following
example:

        CONST FLAG_CH = '^';

The CHR Function

In the 8-bit ASCII set there are numerous character values that
cannot be represented as char constants, since they cannot be
entered from the keyboard when you are using the Editor. To
represent such char values (and for other purposes), Pascal has a
built-in function, CHR. To use this function, the form is

        CHR (expression)

where the expression can be any expression with an integer value
in the range from Ø to 255. Note that this can be just an
integer constant, namely the ASCII code for the desired
character. For instance, the value of

        CHR(Ø)

is the "NUL" (null) character, whose ASCII code number is Ø.

The ASCII codes in the range 128 through 255 are not
assigned to specific characters, but are nevertheless
usable as ASCII code values; thus CHR(2ØØ) is a valid
function call and returns "the character whose ASCII code
is 2ØØ" even though this does not have a standard
interpretation. These characters can be generated on the
keyboard by holding down the Open Apple key and typing a
character. The Open Apple key sets the high bit of the
8-bit ASCII code to 1, which has the effect of adding 128
to the code shown in the ASCII code table in Appendix J.

The CHR function does not check its parameter value to
make sure it is in the range Ø through 255. If the
parameter value is outside this range, CHR will return an
undefined character value.

The Boolean Type

By definition, there are only two boolean values, represented by the words FALSE and TRUE. These words are actually identifiers for built-in constants, whose values represent a logical "false" result and a logical "true" result, respectively. These are not numerical values (though of course they are represented internally as binary numbers). FALSE is less than TRUE.

Boolean values are created in various ways; in particular, the results of comparison operations are boolean values. For example, the value of the expression

    LEGAL_AGE > AGE

(where LEGAL_AGE and AGE are integer variables) is either TRUE or FALSE. An important use for boolean values is in controlling the program. For example, a boolean value can be used to control an IF statement:

    IF LEGAL_AGE > AGE THEN WRITE('Below legal age.')

The WRITE statement is executed if and only if the boolean value of LEGAL_AGE > AGE is TRUE.

Boolean variables are declared by using the reserved word BOOLEAN in the declaration. For example, the declaration

    VAR FLAG1, FLAG2: BOOLEAN;

creates two boolean variables, FLAG1 and FLAG2.

You can also declare boolean constants, by giving FALSE or TRUE as the value.

Defining New Scalar Types

The boolean type is an example of a data type where the values are represented by identifiers. The "meaning" of these values is in the way they are used. You can define a new scalar type by listing the identifiers for its values. For example, the

declaration

        VAR FIDDLE: (BASS, CELLO, VIOLA, VIOLIN);

creates a variable, FIDDLE, whose possible values are the listed
identifiers from BASS to VIOLIN. These four identifiers are
constants. They correspond to the integers Ø, 1, 2, 3, so that
they are strictly ordered; for example, the value CELLO is less
than the value VIOLA.

The syntax diagram for a declaration of a user-defined scalar
type is

user-defined scalar type
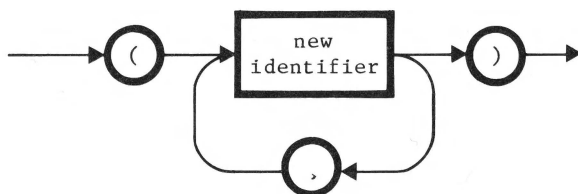
Like boolean values, these user-defined scalar types are useful
for control purposes; examples will be found in Chapter 5.

It is often useful to declare a new type explicitly. For
example, instead of declaring FIDDLE in the manner shown above,
you could first declare the type as follows:

        TYPE STRING_INSTRUMENT = (BASS, CELLO, VIOLA, VIOLIN);

and then declare

        VAR FIDDLE: STRING_INSTRUMENT;

Subrange Types

Subrange types are used to provide automatic run-time range
checking. A subrange type is based upon some scalar type which
is called the "base type." The subrange type is exactly like the
base type, except that its possible values are a subset of the
possible values of the base type. For example, the declaration

VAR X: ∅..255;

creates a variable X which is exactly like an integer variable
except that it can only have values from ∅ to 255. If the
program attempts to give X a value less than ∅ or greater than
255, it will be halted with an error message. Another example:

TYPE CAPITAL_LETTER = 'A' .. 'Z';

This creates the new type CAPITAL_LETTER, whose possible values
are the capital letters from 'A' through 'Z'.

You can create a subrange type based on any scalar type,
including user-defined scalars. For example,

TYPE LOW_STRING = BASS .. VIOLA;

creates the type LOW_STRING, which is a subrange of the type
STRING_INSTRUMENT in a previous example. The syntax for a
subrange type is

subrange type
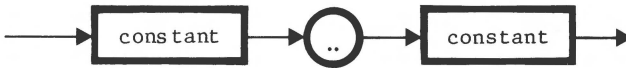
where the two constants must be of the same type (the base
type).

A subrange type is a scalar type. The possible values of a
subrange type cannot be distinguished from the values of the base
type that fall in the same range.

Built-In Functions For Scalar Types

To make good use of the properties of scalar types, Pascal
provides a special set of built-in functions. (Recall that a
function is a subroutine that accepts one or more values as
parameters, and returns one value as a result.) We have already
seen the CHR function, which takes an integer as its parameter
and returns the corresponding character as its result. The other

special functions for scalars are ORD, PRED, and SUCC.

The ORD Function

For any scalar type, each possible value corresponds to a unique
integer. This integer is called the <u>ordinality</u> of the scalar
value. For values of type integer, the ordinality of each value
is the same as the value itself. For all other scalar types, the
ordinalities begin at ∅ for the first value and count up as far
as necessary.

The ORD function accepts any scalar value as its parameter, and
returns the ordinality of that value. The ORD of any single
character is its ASCII code. Examples:

```
ORD(-5) is -5
ORD('A') is 65 (the ASCII code for 'A')
ORD(FALSE) is ∅
ORD(TRUE) is 1
ORD(VIOLA) is 2 (given the declaration shown above)
```

ORD of a boolean value may in some cases return a value
other than ∅ or 1, and should be avoided.

The SUCC and PRED Functions

Within any scalar type, each possible value except the last has a
<u>successor</u>. The successor is simply the next value in sequence,
according to the ordering of the type. Also, each possible value
except the first has a <u>predecessor,</u> which is the value that
precedes it. The SUCC function takes any scalar value and
returns its successor (if it has one). The PRED function takes
any scalar value and returns its predecessor (if it has one).
For example:

```
SUCC(-5) is -4
PRED('c') is 'b'
SUCC(CELLO) is VIOLA (given the declaration shown above)
```

SUCC of the last possible value of a scalar type, or PRED of the first possible value, will return an <u>undefined</u> value. This can cause program bugs. It is <u>up to the</u> program to avoid this situation by checking parameters for SUCC and PRED to make sure that the successor or predecessor exists.

Numeric Functions

A set of simple numeric functions is built into Pascal. Each takes a single numeric value as its parameter, and returns a single value. The first two functions, ROUND and TRUNC, convert real values to integer values. Note that there is no need for a function to convert integer values to real values; they are automatically converted whenever necessary.

The ROUND Function

The ROUND function takes a real value as its parameter and returns an integer value, which is obtained by rounding the real value to the nearest integer. If the parameter is halfway between two integers, then it is rounded away from zero. (Rounding algorithms are discussed in detail in Appendix E.) For example,

```
ROUND(723.3) is 723
ROUND(-5.7) is -6
ROUND(7.7E3) is 7700
ROUND(43.5) is 44
ROUND(-43.5) is -44
```

The TRUNC Function

Like the ROUND function, TRUNC takes a real parameter and returns an integer value; however, the integer is obtained by dropping (truncating) the fractional part of the real value. For example,

```
TRUNC(723.3) is 723
TRUNC(-5.7) is -5
TRUNC(7.7E3) is 7700
```

TRUNC can also accept a long integer value and convert it to an integer value. However, the long integer parameter must not be greater than 32767 or less than -32768.

The ABS Function

The ABS function takes either a real value or an integer value as its parameter, and returns a value of the same type. The value returned is the absolute value of the parameter--that is, if the parameter is negative the sign is changed to positive. For example,

        ABS(3.6545) is 3.6545
        ABS(-3.6545) is 3.6545
        ABS(-512) is 512

The SQR Function

The SQR function takes either a real value or an integer value as its parameter, and returns a value of the same type. The value returned is the square of the parameter.

The ODD Function

The ODD function takes an integer value as its argument, and returns a boolean value. The value returned is TRUE if the integer is odd and FALSE if the integer is even.

Library Functions

A set of trigonometric and exponential functions and a square-root function are provided as a library unit called TRANSCEND. This is described in Appendix A.
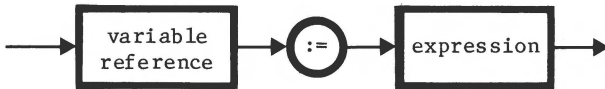
4

Expressions and Assignments

Introduction

In Pascal the value of a variable is changed by the assignment statement. The syntax of an assignment statement is

assignment statement

```
              ┌─────────────┐     ┌───┐     ┌─────────────┐
────────────▶│  variable   │───▶(  :=  )──▶│ expression  │────────▶
              │  reference  │     └───┘     └─────────────┘
              └─────────────┘
```

A variable reference is a reference to a specific, previously declared variable of any type. The syntax is

variable reference

```
  ┌─────────────┐              ┌───┐   ┌─────────────┐   ┌───┐
──│  variable   │──┬────┬────▶(  (  )─┬▶│ expression  │─▶(  )  )─┬──────▶
  │  identifier │  │    │     └───┘ ▲ │ └─────────────┘ └───┘   │
  └─────────────┘  │    │           │ │      ┌───┐              │
                   │    │           │ └─────(  ,  )◀────┘        │
                   │    │           │       └───┘               │
                   │    │     ┌───┐   ┌─────────────┐           │
                   │    └────(  .  )─▶│    field    │───────────┤
                   │          └───┘   │  identifier │           │
                   │                  └─────────────┘           │
                   │          ┌───┐                             │
                   └─────────(  ^  )────────────────────────────┘
                              └───┘
```
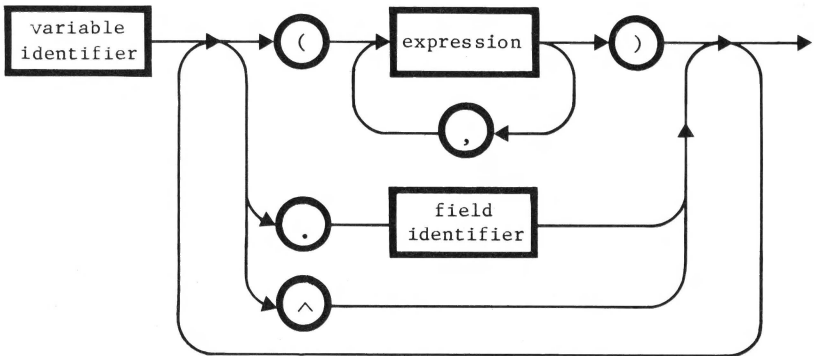
In other words, a variable reference is an identifier which may have a number of "qualifications" appended to it. Each "qualification" is either an array subscript notation in square brackets, a record field identifier set off by a period, or the ^ symbol to indicate the object of a pointer. Arrays, records, and pointers are fully explained in other chapters.

The symbol := is known as the assignment operator. It means that the expression on the right-hand side of the assignment operator is to be evaluated, and the result is to become the new value of the variable that is referred to on the left-hand side.

In its simplest form, an expression may be a constant, a variable reference, or a function call. More generally, an expression is a combination of operands and operators.

An operand is a single value, such as a constant, variable
reference, or function call. Two operands can be combined by
means of an operator, such as *, +, -, or <=. Examples:

        X <= XLIMIT
        RAINFALL.OLDVAL + INCREMENT
        MARGIN[INDEX] - 3

There are also operators that take only one operand: the NOT
operator takes a single operand, and the + and - operators may
take a single operand. Examples:

        -X
        NOT TSTRESULT

The simplest form of an operand is a variable reference or a
constant. Examples of assignments which use operators that take
two operands are:

        RESULT := ONEOPERAND * ANOTHEROPERAND;
        TRUTH := BIGINTEGER >= LITTLEINTEGER;
        FALSITY := MYDOG > YOURDOG;

The operands in an expression can be much more complex than this;
in fact, an operand itself can be an expression (which may be
enclosed in parentheses, depending on context). Notice that this
allows expressions to be complicated nestings of operations.
Some slightly more sophisticated assignment statements are:

        CHAMBER.PRESSURE := N * R * CHAMBER.TEMP / CHAMBER.VOL;
        FINISHED := (TEST = TRUE) AND (ERROR = FALSE);
        ANSWER[K] := TRUNC(SQR(COS(THETA[K])));

Precedence of Operators

Pascal expressions often contain more than one operator. The
rules of precedence determine the way in which the operations
within the expression are grouped. If there were no precedence,
the expression:

        A - B * C / D + E

would be evaluated from left to right as if it read:

    (((A - B) * C) / D) + E

But the operators do have precedence. For example, the * and /
operators are always applied before the + and - operators,
regardless of the sequence in the expression (unless parentheses
are used to overrule the precedence). The above example is
actually evaluated as if it read:

    (A - ((B * C) / D)) + E

The operations are grouped this way because * and / have higher
precedence while + and - have lower precedence.

Pascal has four different levels of operator precedence, as shown
in the following table:

ORDER OF PRECEDENCE	OPERATORS	DESCRIPTION
1	NOT	NOT operator
2	* / DIV MOD AND	Multiplying operators
3	+ - OR	Adding operators
4	= <> < <= >= > IN	Relational operators

In this table, each line contains operators of equal precedence.
The NOT operator has the highest precedence.

In an expression all of the operators of the highest level are
applied before any of the operators of the next level are
applied. If an expression contains more than one operator of the
same precedence they are applied from left to right.

The precedence table is one of the significant differences
between Pascal and many other languages. For example, in
many languages AND and OR have the same precedence. Also,
in Pascal each operator symbol always has the same
precedence even if it has two possible meanings; e.g. the
+ operator can mean arithmetic addition or set union
(depending on its operands) but it has the same precedence
in either case. The virtue of Pascal's precedence table
is that it is extremely easy to remember.

A portion of an expression that is enclosed in parentheses is
called a subexpression. A subexpression is evaluated as if it
were an independent expression, before it is combined with any
other parts of the expression. If there are nested parentheses

(parentheses within parentheses) in the expression, the innermost
subexpression is evaluated first. Parentheses can be used, as in
ordinary algebra, to override the precedence of operators.

Here are a few examples of how expressions are evaluated:

	EXPRESSION	EVALUATED AS	RESULT
1.	4 + 3 * 2 - 1	(4 + (3 * 2)) - 1	9
2.	8 * 2 > 5 + 6	(8 * 2) > (5 + 6)	TRUE
3.	2 > 1 AND 4 < 5	(2 > (1 AND 4)) < 5	illegal!
4.	5 MOD 4 + 3	(5 MOD 4) + 3	4

In expression 1, the multiplying operator * is applied first, as
if the expression were written 4 + (3 * 2) - 1. Since + and -
are both of the same precedence, the remaining expression is
evaluated from left to right, giving 9 as result.

Expression 2 is a relational operation between two arithmetic
operations. The * operator is applied first, then the +
operator, then the > operator. Because 8 * 2 is greater than
5 + 6, the result of the entire expression is TRUE.

Expression 3 appears to be a boolean operation between the
results of two relational operations. The highest
precedence operator, however, is AND which is applied to 1
and 4. Because AND requires boolean operands, this
expression is ILLEGAL. It could be properly written as
(2 > 1) AND (4 < 5).

Expression 4 is evaluated exactly as it is written.

If a function call appears in an expression, the function is
called, and a value is returned, before that value is used as an
operand of some operator.

The precedence rules specify the order of performance of
operations. However, do NOT make any assumptions about
the order of evaluation of operands.

The Arithmetic Operators

The arithmetic operators, used with operands of type integer, long integer, and real, are:

*	multiplication
/	division with real result
DIV	division of integers with integer result
MOD	remainder of integer division
+	addition
-	subtraction or negation

Each operator has specific rules concerning what type of operands it may take, and what type of results are produced. A universal rule is that no arithmetic operator may be used to combine a real operand with a long integer operand.

In Apple III Pascal, the arithmetic operations on real values conform to the proposed IEEE floating-point standard. Under default conditions, these operations behave in ways that are familiar to most programmers. However, some subtle variations in the execution of operations on real values are available by methods described in Appendix E.

Remember that each type of result has size constraints. An integer result must fall within the range -32768 to 32767 while a long integer result cannot exceed 36 decimal digits (not including sign). A real result must have an absolute value in the range 1.4Ø1298464E-45 to 3.4Ø2823466E38, or Ø.Ø.

Integer arithmetic overflow occurs when an integer arithmetic operation results in a final or intermediate result outside the limits given above. This does not cause an error halt; instead it produces an undefined integer result. It is up to the programmer to be sure that integer arithmetic will not overflow.

Real arithmetic overflow occurs when a real arithmetic operation would result in a final or intermediate result

with an absolute value greater than the upper limit given
above. This causes an error halt under default
conditions; see Appendix E for details. A real result
with an absolute value less than the lower limit does not
cause an error; it produces a result of 0.0.

The * Operator

Multiplication is done with the * operator. The results of
multiplication operations are as follows:

MULTIPLICAND	MULTIPLIER	RESULT TYPE
integer	integer	integer
long integer	long integer	long integer
real	real	real
integer	long integer	long integer
integer	real	real
real	long integer	not allowed

Two examples of multiplication in expressions are:

```
writeln('two cubed = ', 2*2*2);
A := C * D
```

The / Operator

The / operator is used for division and it always gives real
results. The results of division operations are:

DIVIDEND	DIVISOR	RESULT TYPE
integer	integer	real
real	real	real
integer	real	real
real	integer	real
long integer	anything	not allowed
anything	long integer	not allowed

Example:

```
APPROXPI := 355 / 113;    { APPROXPI is real }
```

The DIV Operator

DIV is the integer division operator. It is used with integer
and long integer operands with the following results:

DIVIDEND	DIVISOR	RESULT TYPE
integer	integer	integer
long integer	long integer	long integer
integer	long integer	long integer
long integer	integer	long integer
real	anything	not allowed
anything	real	not allowed

The DIV operation produces a result that is truncated toward $\emptyset$: the remainder of the division is lost. The expression

    A DIV B

(where A and B are integer values) is equivalent to

    TRUNC(A / B)

This shows the relationship between DIV and /.

The MOD Operator

MOD takes two integer operands and returns an integer value that is the remainder of the absolute value of the first operand divided by the absolute value of the second operand.

A typical application of the MOD function is

    A MOD B = $\emptyset$

which is TRUE if B is a factor of A.

This implementation of the MOD operator does not correspond to the definition of MOD given by Jensen and Wirth.

The + Operator

The + operator is used for addition of integers, long integers and reals. Results of addition operations are:

OPERAND TYPES		RESULT TYPE
integer	integer	integer
long integer	long integer	long integer
real	real	real
integer	long integer	long integer
integer	real	real
real	long integer	not allowed

Example:

```
XREAL := AINTEG + XREAL + 2;
```

The + operator can be used with a single operand as a sign indicator. This use of + produces a result that is identical to its operand.

The — Operator

The - operator is used for subtractions of integer, long integer and real operands. Operations using the - operator give these results:

MINUEND	SUBTRAHEND	RESULT TYPE
integer	integer	integer
long integer	long integer	long integer
real	real	real
integer	long integer	long integer
long integer	integer	long integer
integer	real	real
real	integer	real
real	long integer	not allowed
long integer	real	not allowed

Example:

```
FRACTION := A / B - A DIV B;   {A,B:integer, FRACTION:real}
```

The - operator can also be used with a single operand (integer, long integer, or real) to perform arithmetic negation, i.e. to change the sign of the operand. Examples of negation are:

```
-3
-A
-TRUNC(A)
-(3 + 2)
-(-2)
```

If a negated operand is used after one of the arithmetic
multiplying operators (*, /, DIV, or MOD) the operand must be
enclosed in parentheses. Two examples are:

    -A * (-B)
    4 * (-OFFSET) MOD (-(3 + 2))

The Relational Operators

The relational operators are used to make comparisons between
scalar and/or real operands. They are especially useful with the
flow of control statements which are explained in the next
chapter. The relational operators are:

>	greater than
>=	greater or equal to
=	equal to
<	less than
<=	less than or equal to
<>	less than or greater than

Another relational operator, IN, is used with set values. It is
explained in Chapter 7.

Comparisons are always made between two values of the same scalar
or numeric type, and yield boolean results. If an integer is
compared with a real, the integer is first converted to a real,
and then a real-to-real comparison is done. Likewise, a
comparison between an integer and a long integer will convert the
integer to a long integer before invoking a long integer
comparison. No other implicit type conversions are done by these
operators; in general, a comparison that mixes types will be
flagged by the Compiler. Boolean and user defined scalars may
only be compared with values of the same type. Of course
expressions may also be compared, provided that the comparison
follows the rules described above. Here are some examples of
properly used relational operators:

    IF SCORES[I] > MAXSCORE THEN MAXSCORE := SCORES[I];
    WHILE ANGLE < CIRCUMFERENCE / ARCLENGTH * 36Ø DO...;
    REPEAT
        ...
    UNTIL INDEX > LIMIT

Notice that relational operators, because they give boolean
results, are a primary tool used to control program flow. The
use of boolean values in IF, WHILE and REPEAT statements is
explained in the next chapter.

User-defined scalars can be compared using any of the relational
operators. The ordinality of a value of a user-defined scalar
type is determined by its position in the type's declaration.
Thus, with the declaration

    VAR PAINT: (NONE, RED, ORANGE, YELLOW);

the statement

    IF PAINT > NONE THEN PAINTPICTURE;

will cause the procedure PAINTPICTURE to be executed if the value
of PAINT is RED, ORANGE, or YELLOW.

The result of comparing two real values will be exactly one of
four possible relations:

    (1) equal,
    (2) less than,
    (3) greater than, or
    (4) unordered.

We are used to thinking that if two numbers are unequal, then one
must be larger than the other. But the real type includes, in
addition to numeric values, special diagnostic values that result
from invalid operations and other special events (like division
by zero). These diagnostic values may compare "unordered" with
numeric values, and such comparisons may even cause runtime
halts. See Appendix E for details; for ordinary programming just
remember that, for real comparisons,

    - "a <> b" [that is, a less than or greater than b] is
      not synonymous with "not (a = b)", and

    - if "a < b" is false, you don't automatically know that
      "a >= b", because a and b may be unordered with respect
      to one another.

The relational operators can also be used to compare structured
types. The operations permitted between operands of the
structured types (arrays, sets, strings and records) are
explained as each structured type is explained.

Logical Operators

Sometimes you need to find a result that is dependent on more
than one boolean value. In Pascal this capability is provided by
the logical operators which, in order of precedence, are:

      NOT   boolean negation
      AND   boolean conjunction
      OR   boolean disjunction

Logical operators take boolean operands and produce boolean
results according to the following rules:

A	B	A AND B	A OR B	NOT B
TRUE	TRUE	TRUE	TRUE	FALSE
TRUE	FALSE	FALSE	TRUE	TRUE
FALSE	TRUE	FALSE	TRUE	FALSE
FALSE	FALSE	FALSE	FALSE	TRUE

As you can see the operator NOT takes a single boolean operand
whereas the operators AND and OR both require two boolean
operands.

Examples of correctly used logical operators are:

      (A > B) AND (C < SQR(D))   {SQR(D) returns square of D}
      A AND B AND NOT(A AND B)   {result is always false}
      (COUNT <= 1ØØ) OR ERROR   {ERROR is a boolean variable}

Although it is not always necessary to evaluate both
operands of a boolean expression in order to determine the
result, a program may nevertheless evaluate both
operands.

Relational Operators with Boolean Operands

Each of the relational operators (=, <>, <=, <, >, >=, IN) yields
a boolean value. Furthermore, the type boolean is defined such
that FALSE < TRUE. Therefore, it is possible to define each of
the 16 boolean operations using logical and relational
operators. If p and q are boolean values, one can express

      implication   as   p <= q
      equivalence   as   p = q
      exclusive OR as   p <> q

Note that without the use of relational operators you would have
to express the exclusive OR function as

      (p AND NOT q) OR (NOT p AND q)

Result Types

The result types for all combinations of operator and operands
are described in full detail in the preceding sections. This
section summarizes those descriptions.

In the table of results below, the column on the right lists the
operators that may not be used with the pair of operands in that
row.

OPERAND TYPES		RESULT TYPE	ILLEGAL OPERATORS
integer	integer	integer (real for / operator)	
long integer	long integer	long integer	/, MOD
real	real	real	DIV, MOD
integer	real	real	DIV, MOD
integer	long integer	long integer	/
real	long integer	not allowed	ALL

The logical operators NOT, AND and OR all take boolean operands
and give boolean results. NOT precedes its single operand; AND
and OR each take two operands.

The relational operators >, >=, =, <=, < and <> can be used to compare two boolean values, two user defined scalar values of the same type, or two numeric values. The only numeric values that cannot be compared are a real with a long integer. All relational operations yield boolean results.

Assignments

There are also restrictions concerning what types of values may be assigned to what types of variables. The legal assignments for non-structured variables are:

VARIABLE TYPE		EXPRESSION TYPE
integer	:=	integer
long integer	:=	integer or long integer
real	:=	integer or real
boolean	:=	boolean
char	:=	char

To do assignments that are not shown in this table, use type conversion functions. These are described in Chapter 3.

5

The Flow of Control

Introduction

We begin with a brief digression on Pascal statements in general, since this chapter describes most of the types of statements. There are ten kinds of statements in Pascal; all are introduced in Chapter 2. Thus we have

statement

```
           ┌──────────────────────┐
        ┌─→│ assignment statement  │─┐
        │  └──────────────────────┘ │
        │  ┌──────────────────────┐ │
        ├─→│  compound statement   │─┤
        │  └──────────────────────┘ │
        │  ┌──────────────────────┐ │
        ├─→│    procedure call     │─┤
        │  └──────────────────────┘ │
        │  ┌──────────────────────┐ │
        ├─→│    for statement      │─┤
        │  └──────────────────────┘ │
        │  ┌──────────────────────┐ │
        ├─→│   repeat statement    │─┤
        │  └──────────────────────┘ │
        │  ┌──────────────────────┐ │
        ├─→│   while statement     │─┤
        │  └──────────────────────┘ │
        │  ┌──────────────────────┐ │
        ├─→│     if statement      │─┤
        │  └──────────────────────┘ │
        │  ┌──────────────────────┐ │
        ├─→│    case statement     │─┤
        │  └──────────────────────┘ │
        │  ┌──────────────────────┐ │
        ├─→│    goto statement     │─┤
        │  └──────────────────────┘ │
        │  ┌──────────────────────┐ │
        └─→│    with statement     │─┤
           └──────────────────────┘ │
                                     └──→
```
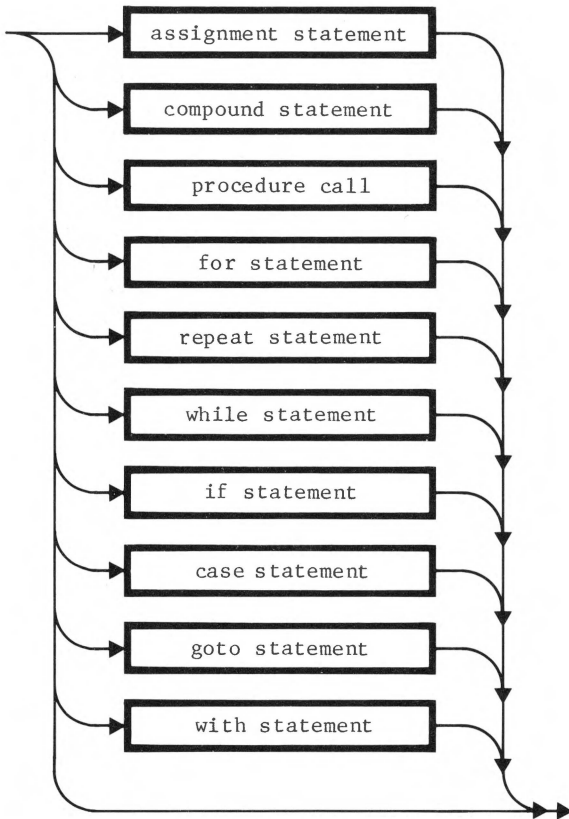
The assignment statement is described in Chapter 4. The WITH statement is described in Chapter 8.

Notice that technically, there is an eleventh type of
statement which consists of nothing; this is called a
"null statement." It simply means that whenever Pascal
syntax calls for a statement, you can omit it.

It also means that when a program contains an unnecessary
semicolon, the Pascal Compiler considers the semicolon to
be separating a null statement from another statement.
The result is two statements where you intend to have only
one. Most of the time this is harmless, but occasionally
it causes a compilation error because only one statement
is allowed.

Except for the WITH and assignment statements, all the statements
shown in the diagram are statements that determine the flow of
control in a Pascal program; that is, they determine the order in
which other statements are executed. There are "flow of control"
statements that repetitively or conditionally execute other
statements, and there are statements that transfer control to
another portion of the program.

The flow of control statements can be subdivided into groups.

- The compound statement groups several statements into
  one.

- The procedure call statement causes execution of a
  procedure.

- The repetition statements (FOR, REPEAT and WHILE) allow
  a sequence of statements to be executed repeatedly.

- The conditional statements (IF and CASE) permit
  conditional execution of statements.

- The GOTO statement permits unconditional transfer of
  control from one part of the program to another.

The Compound Statement

The compound statement is one of the most useful statements in
Pascal. The syntax is

compound statement
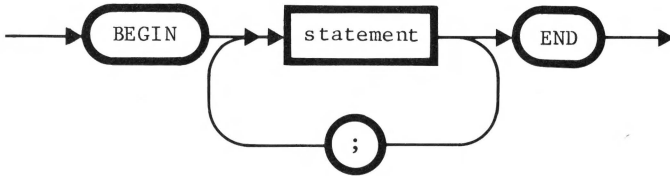
A compound statement can contain any number of statements of any type, separated by semicolons. Note that the word BEGIN does not have a semicolon after it, since it is not a statement; likewise, there is no semicolon just before the word END because END is not a statement.

The compound statement is always considered as a single statement, even though it may contain more than one statement. Keep this in mind since most of the other flow of control statements act on single statements. Whenever you want a sequence of statements to be treated as a single statement, simply delimit it with BEGIN and END.

The Procedure Call

A procedure is called by merely mentioning its name, with whatever parameters the procedure may require. The syntax for a procedure call is

procedure call
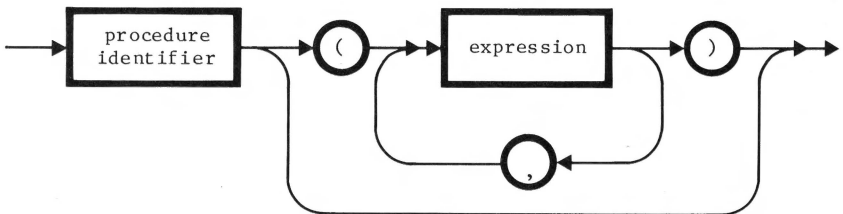
The procedure identifier is the name of the procedure to be called. The list of expressions in parentheses is called the parameter list of the procedure call. The number of parameters in the list depends on the procedure being called.

The effect of a procedure call is to execute the procedure immediately (passing the specified parameters, if any). When the procedure terminates, control is transferred to the statement following the procedure call. See Chapter 6 for complete information on procedures.

The Repetition Statements

Pascal has three statements that can repeatedly execute a statement or sequence of statements. Termination of the repetition is determined by the state of control variables or expressions. The repetition statements are the FOR, REPEAT and WHILE statements.

The FOR Statement

The FOR statement is used to execute a statement a specific number of times. This is done by executing the statement once for each value of a control variable between an initial value and a limit value. The syntax of the FOR statement is

for statement
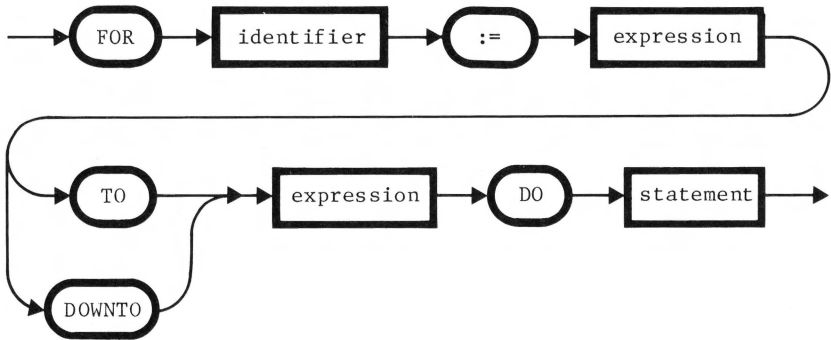
where the identifier is the identifier of a variable called the control variable. The control variable may be of any scalar type. The two expressions must be of the same scalar type as the control variable.

The control variable must be a simple variable; it cannot be an array element, a record field, or a dynamic variable.

The value of the "initial" expression is called the initial value, and the value of the "limit" expression is called the limit value.

For example, suppose that we have an array of 24 integer values and an integer value that can be used to index it:

```
VAR VAL: ARRAY [1..24] OF INTEGER;
    IX:  INTEGER;
```

Now suppose that we want to multiply each value in the array by 2. We can do this with a FOR statement:

```
FOR IX := 1 TO 24 DO VAL[IX] := 2*VAL[IX]
```

Note that the control variable is available within the FOR statement. (Remember that the control variable, like any other variable, must be declared.) However, the value of the control variable should not be changed within the FOR statement.

If the value of the control variable is changed within the FOR statement, the results are unspecified as this is a violation of the rules of Standard Pascal.

In the above example, the embedded assignment statement is executed 24 times. Each time, the control variable (IX) takes on a new value; this value is 1 the first time and 24 the last time.

If we want to write out each value after multiplying by 2, we can use a compound statement inside the FOR statement:

```
FOR IX := 1 TO 24 DO BEGIN
                      VAL[IX] := 2*VAL[IX];
                      WRITELN(VAL[IX])
                     END
```

Processing of the FOR statement (using TO instead of DOWNTO) is as follows:

- First, the initial value is calculated (just once) and assigned to the control variable.

- Then, the limit value is calculated (just once).

- If the initial value is greater than the limit value, the remainder of the FOR statement is skipped. Otherwise, the following steps are taken:

    1. The statement following the word DO is executed.

    2. The control variable is assigned the value of its own successor.

    3. If the new value of the control variable is not greater than the limit value, go back to Step 1 and repeat. Repetition continues until the value of the control variable is greater than the limit value.

After the FOR statement has finished executing, the value of the control variable is unspecified.

The DOWNTO option of the FOR statement sets the control variable to its own <u>predecessor</u> after each iteration, stopping when the value of the control variable is <u>less</u> than the limit value.

Some points to remember about Pascal FOR statements:

- A variable of any scalar type (integer, char, boolean, user-defined, or subrange) may be used as a control variable. However, a control variable of type real is not allowed.

- If the value of the control variable is changed within the FOR statement, the results are unspecified.

- Use of the control variable in either limit expression produces undefined results.

The REPEAT Statement

The REPEAT statement, like the FOR statement, is used to control repetition in a program. The syntax is

repeat statement

```
───▶( REPEAT )──┬─▶│ statement │──┬─▶( UNTIL )──▶│ expression │───▶
                │                  │
                └──────◀──( ; )──◀─┘
```
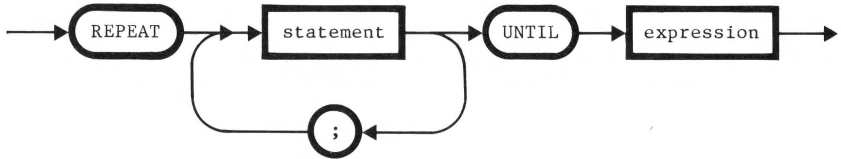
Note the differences from the FOR statement. The sequence of
statements in a REPEAT statement doesn't need to be delimited by
a BEGIN and an END; the REPEAT and UNTIL do this. The expression
after UNTIL must have a boolean result. It is evaluated after
each execution of the enclosed statements; hence the statements
are always executed at least once. Notice that if the expression
is never TRUE, the statements will be repeated forever.

A typical application of the REPEAT statement is

```
REPEAT
  WRITE('ENTER A NUMBER BETWEEN Ø AND 1ØØ -> ');
  READLN(INTVAR);
  WRITELN('2 times ', INTVAR, ' is ', 2*INTVAR)
UNTIL (INTVAR < Ø) OR (INTVAR > 1ØØ)
```

This REPEAT statement executes the contained statements until the
user types a number less than Ø or greater than 1ØØ.

The WHILE Statement

The WHILE statement is similar to the REPEAT statement. The
syntax is

while statement

```
───▶( while )──▶│ expression │──▶( DO )──▶│ statement │───▶
```
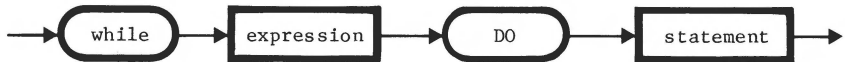
Unlike the REPEAT statement, the WHILE statement evaluates the
controlling expression <u>before</u> each repetition of its statement.

If the expression, which must have a boolean result, is initially false, the statement will not be executed.

The WHILE statement acts on a single statement; thus a compound statement must be used if more than one statement is to be executed following the word DO.

For example, suppose that a program requires the user to type a number from 1 to 8. If the user does this, the program can continue. But if the user types a number that is out of range, the program must display an error message and give the user another chance; and this should be repeated until the user types a number in the required range. This is a natural application for the WHILE statement:

```
WRITE('Type a number from 1 to 8: ');
READLN(NUMBER);
WHILE (NUMBER < 1) OR (NUMBER > 8) DO BEGIN
  WRITELN('Number must not be less than 1 or more than 8!');
  WRITE('Try again. Type a number from 1 to 8: ');
  READLN(NUMBER)
END
```

If the user gives a correct response the first time, the WHILE statement is never executed. But if the user's response is out of range, the WHILE statement executes repeatedly until the user responds correctly. In either case, the value of NUMBER is guaranteed to be in the range 1..8 at the end of this sequence.

The Conditional Statements

The IF statement and the CASE statement are used to execute a statement if a variable or expression has a desired value.

The IF Statement

The IF statement contains a boolean expression, a statement to be executed if the value of the expression is TRUE, and (optionally) another statement to be executed if the value of the expression is FALSE. The syntax of the IF statement is

if statement

```
──►──( IF )──►──[ expression ]──►──( THEN )──►──[ statement ]──┬──►──( ELSE )──►──[ statement ]──┬──►►
                                                               └──────────────────────────────────┘
```
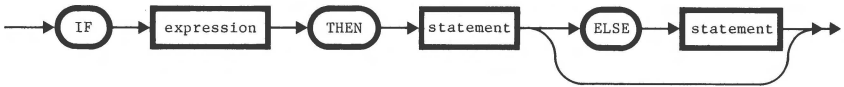
When an IF statement is executed, the following sequence of
events takes place.

- The boolean expression is evaluated.
- If the boolean expression is TRUE:
    - The statement following THEN is executed.
- If the boolean expression is FALSE:
    - If there is an ELSE, the statement following
      ELSE is executed.
    - If there is no ELSE, the IF statement has no
      effect.

Note that just one statement is allowed after the word THEN; it
may be a compound statement. Likewise, just one statement
(possibly compound) is allowed after the word ELSE. Here is an
example of an IF statement without an ELSE part:

```
IF TOTAL > 1ØØ THEN BEGIN
                    WRITELN('Error:  Total too big');
                    TOTAL := TOTAL - CRNT
                END
```

If the value of the variable TOTAL is greater than 1ØØ, the
compound statement is executed to display the message
"Error: Total too big" and adjust the value of TOTAL; otherwise
the compound statement is not executed.

The ELSE part of an IF statement is only executed if the result
of the boolean expression is false. For example, the statement

```
IF TOTAL > 1ØØ THEN BEGIN
                    WRITELN('Error:  Total too big');
                    TOTAL := TOTAL - CRNT
                END
            ELSE WRITELN('Total is ', TOTAL)
```

will execute the compound statement if the value of TOTAL is
greater than 1ØØ (just as in the previous example); but if the
value of TOTAL is not greater than 1ØØ, then the WRITELN
statement following the word ELSE is executed to display the
message "Total is " followed by the value of TOTAL.

It is important to be careful of semicolon placement in IF
statements. For example

```
IF A = B THEN BEGIN
                WRITELN('A equals B');
                EQCOUNT := EQCOUNT + 1
              END
         ELSE WRITELN('A not equal to B')
```

is correct: there is no semicolon after the END because
the ELSE does not start a new statement--it is a
continuation of the same IF statement. A common mistake
is to put a semicolon before the ELSE, which causes a
compilation error because there is no Pascal statement
that begins with the word ELSE.

Nested IF Statements

The statement following the word ELSE can be an IF statement, and
can contain its own ELSE clause. Thus a statement can be written
to take different actions for each of several mutually exclusive
conditions.

```
REPEAT
  WRITE('Enter command S,D,P,Q,E -> ');
  READLN(COMM);
  IF COMM = 'S' THEN SHUFFLEDECK
    ELSE IF COMM = 'D' THEN DEALCARDS
      ELSE IF COMM = 'P' THEN DISPLAYPOINTS
        ELSE IF (COMM = 'Q') OR (COMM = 'E') THEN QUIT
UNTIL (COMM='Q') OR (COMM='E')
```

Conditions will only be checked until a true one is found. The
greatest efficiency is achieved if the most probable conditions
are checked first.

The statement following the word THEN can also be a nested IF
statement, but this construction is less useful and can lead to a
confusing program. Be careful with the following type of

construction:

```
IF A=B THEN IF C=D THEN WRITELN('A=B and C=D')
                   ELSE WRITELN('A=B but C<>D')
```

The ELSE matches the last preceding THEN, as indicated by the indentation.   If you add another ELSE it will match the first THEN:

```
IF A=B THEN IF C=D THEN WRITELN('A=B and C=D')
                   ELSE WRITELN('A=B but C<>D')
       ELSE WRITELN('A<>B')
```

The above statement can be clarified, without changing its meaning, by making the nested statement a compound statement:

```
IF A=B THEN BEGIN
            IF C=D THEN WRITELN('A=B and C=D')
                   ELSE WRITELN('A=B but C<>D')
            END
       ELSE WRITELN('A<>B')
```

Now it is obvious which ELSE matches which THEN.

The CASE Statement

The CASE statement uses the value of an expression to select and execute one statement from a list of statements.   The controlling expression and the list of statements are contained within the CASE statements, and each statement in the list is "labeled" with one or more constants called case selectors which are possible values of the controlling expression.   An OTHERWISE clause is optional; if present, it contains a statement which is executed if the controlling expression's value does not match any of the case selectors.

The syntax is

case statement
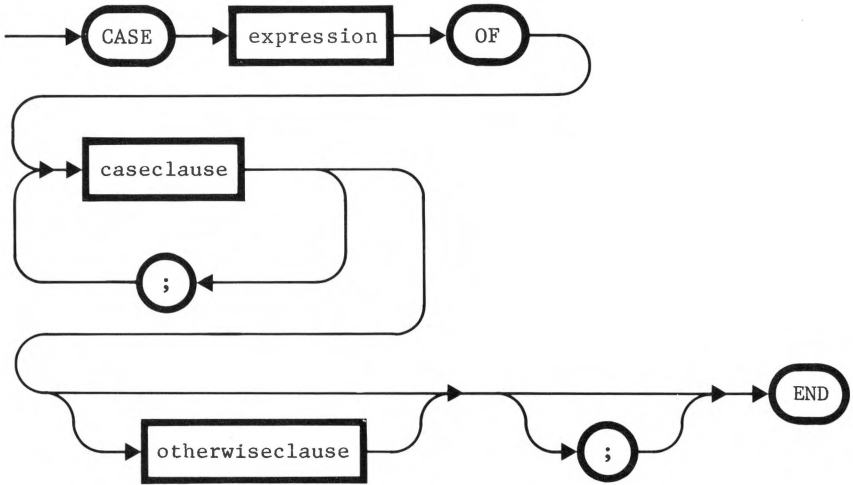
caseclause
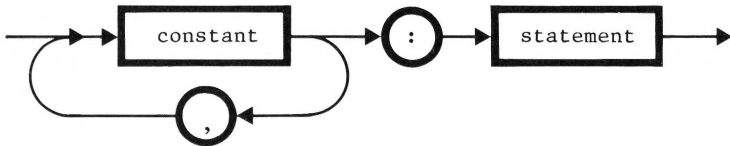
otherwiseclause
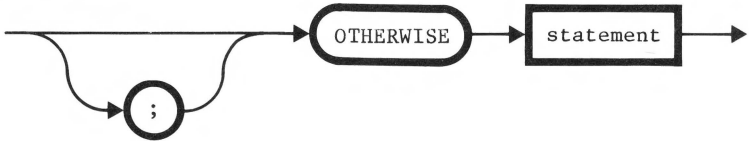
where the expression must give a result of a scalar type, and the
constants in each caseclause must be of the same type. The
expression is evaluated and the result is sequentially compared
with the constants in each caseclause. If the result matches one
of the constants, only the statement in that caseclause is
executed.

If no match is found and there is an OTHERWISE clause, the
statement in the OTHERWISE clause is executed. If no match is
found and there is no OTHERWISE clause, then the CASE statement
has no effect.

In discussing nested IF statements above, we gave the following
example:

```
    REPEAT
      WRITE('Enter command S,D,P,Q,E -> ');
      READLN(COMM);
      IF COMM = 'S' THEN SHUFFLEDECK
        ELSE IF COMM = 'D' THEN DEALCARDS
          ELSE IF COMM = 'P' THEN DISPLAYPOINTS
            ELSE IF (COMM = 'Q') OR (COMM = 'E') THEN QUIT
    UNTIL (COMM='Q') OR (COMM='E')
```

Exactly the same effect can be achieved more naturally with a
CASE statement:

```
REPEAT
  WRITE('Enter command S,D,P,Q,E -> ');
  READLN(COMM);
  CASE COMM OF
    'S': SHUFFLEDECK;
    'D': DEALCARDS;
    'P': DISPLAYPOINTS;
    'Q', 'E': QUIT
  END
UNTIL (COMM='Q') OR (COMM='E')
```

If we used the nested IF statement and tried to allow for the possibility of lower-case letters, the resulting nest would be unwieldy. With a CASE statement, however, the enhancement is easy. We also add an OTHERWISE clause to handle invalid command input by calling a procedure named HELP.

```
REPEAT
  WRITE('Enter command S,D,P,Q,E -> ');
  READLN(COMM);
  CASE COMM OF
    'S', 's': SHUFFLEDECK;
    'D', 'd': DEALCARDS;
    'P', 'p': DISPLAYPOINTS;
    'Q', 'E', 'q', 'e': QUIT
    OTHERWISE HELP
  END
UNTIL (COMM='Q') OR (COMM='E') OR (COMM='q') OR (COMM='e')
```

As in all other cases where only a single statement is allowed, each statement within a caseclause or OTHERWISE clause can be a compound statement.

Caution is required if you use integer constants as case selectors in a CASE statement. If the difference between the largest and the smallest case selector in a CASE statement is too great, the Compiler will be unable to compile it. The limit depends on the statements within the CASE statement, but as a rule of thumb do not use any integer case selectors that differ by more than 100.

The reason for this is that to implement a CASE statement, the Compiler builds a table in the code with an entry for each possible case selector from the smallest actually used to the largest.

The EXIT and HALT Procedures

The Pascal statements already described should be adequate for
controlling the flow of almost all programs. In some cases, it
is convenient to be able to exit immediately. The EXIT procedure
allows this; the syntax for calling EXIT is

```
──►─( EXIT )──►─( ( )──┬──►─[ identifier ]──┬──►─( ) )──►──
                       └──►─( PROGRAM )──────┘
```
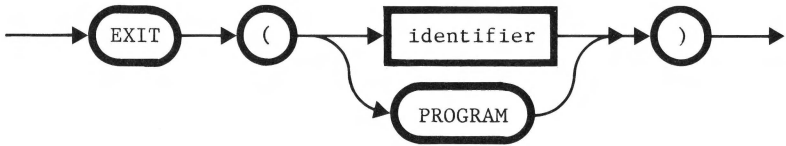
where the identifier is the name of a procedure, a function, or
the program. The most common use of EXIT is with either the word
PROGRAM or the program name as the parameter; EXIT then
terminates the program in an orderly manner: all open files are
closed (see Chapter 1∅), and control returns to the command level
of the system just as if the program had reached its END.

When a procedure or function identifier is used in calling EXIT,
the specified procedure or function is exited.

Do not use this technique unless you are sure you
understand what you are doing (Chapter 6 covers procedures
and functions).

Control returns to the point where the procedure or function was
called, just as if the procedure or function had reached its END;
however, if a function is exited without an assignment being made
to the function identifier, the function will return an
unpredictable value.

Note that the specified procedure or function need not be the one
in which the EXIT procedure is called. EXIT follows the trail of
procedure calls back to the procedure or function specified; each
procedure or function in the trail is exited (whether or not it
has completed its execution). If the specified procedure or

function is recursive (see Chapter 6), then the most recent
incarnation is exited; earlier incarnations will complete their
executions normally.

The HALT procedure takes no parameters. It brings the program to
an immediate halt with a non-fatal run-time error.

The GOTO Statement

Some programming situations demand an instant transfer of control
in a manner that is not easy to achieve using the repetition or
conditional statements. To handle these situations Pascal has
the GOTO statement. The GOTO statement should be used only in
those unusual cases that cannot be handled easily by the other
control statements. By default, the Compiler does not allow GOTO
statements. If your program uses GOTO statements you must
include the Compiler option

    {$GOTO+}

This option must precede the first GOTO statement in the
program. Please see Appendix F for more information about
Compiler options.

The GOTO statement causes a direct transfer of control to a
labeled statement that is in the same procedure or function as
the GOTO statement (considering the main program to be a
procedure). The extremely uncomplicated syntax of GOTO is

goto statement

where the label is an unsigned integer of not more than four
digits. The label must first be declared. Label declarations
come immediately after the heading of a program, procedure, or
function, before any other declarations. The following program,
which loops infinitely, shows legal use of label declarations and

the GOTO statement. It also shows some of the problems of the
GOTO statement.

```
{$GOTO+}
PROGRAM JUMP;
LABEL 1, 5326, 42, 999;
BEGIN
  1: GOTO 5326;
  999: GOTO 42;
  5326: GOTO 999;
  42:  GOTO 1
END.
```

Note that you cannot jump out of a block with a GOTO
statement.

Passing control to a statement that is inside a structured
statement from a point outside the structured statement by
means of a GOTO has underlined effects, although the
Compiler will not indicate an error. All of the GOTO
statements in the following example are wrong for this
reason.

```
IF TRUE THEN
  123: GOTO 6;
FOR INDEX := 1 TO 10 DO
  6:   GOTO 123;
BEGIN
  1: WRITELN('GOTO ANOTHER PROCEDURE');
  GOTO 6
END;
GOTO 1
```

6

Procedures and Functions

Introduction

Procedures and functions are the subroutines of Pascal. Each procedure or function is a distinct section of code, contained within a program, that is executed when the program <u>calls</u> it. In many cases the program calls it more than once.

A procedure can be thought of as a subprogram nested in the main program (or within another procedure, or within a function). Just as you define a Pascal program by writing it in text form, you define a procedure by writing a <u>procedure definition</u> into the text of the program. If you study the syntax diagrams further on in this chapter you can see that a procedure definition, like a program, contains one <u>block</u>. The block may contain other procedure definitions. Thus procedures (and functions) can be freely nested within each other. Indeed, for purposes of program execution the system considers the program itself to be just the outermost procedure of a nested structure of procedures and functions.

A procedure is called by means of a procedure call statement, which refers to the procedure by name and supplies values for any <u>parameters</u> belonging to the procedure. Parameters are a special kind of variable used to pass information to the procedure when it is called; they are discussed in detail below.

A function is similar to a procedure except that it is called by means of a function reference instead of a call statement. The function reference appears in an expression; it references the function by name and supplies any parameters required by the function. The function returns a value; that is, it computes a value, and this value replaces the function reference when the expression is evaluated.

Procedures and functions are defined (written) after the variable declarations, if any, and before the compound statement that contains the statements of the program.

Defining a Procedure

A procedure definition consists of a procedure heading, a block,
and a terminating semicolon:

procedure definition
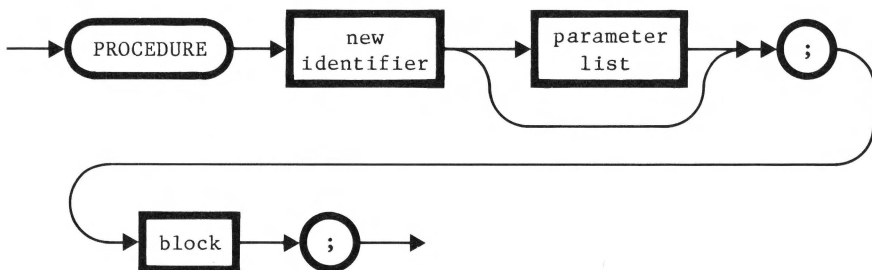
The first part of the procedure definition--the word PROCEDURE,
the identifier, the parameter list, and the semicolon--are called
the procedure heading. The remaining part--the block and the
final semicolon--are called the procedure body. The syntax for a
parameter list is

parameter list
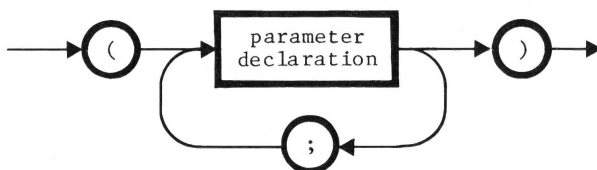
The parameter list declares the procedure's parameters, if any.
It consists of an opening parenthesis, one or more parameter
declarations separated by semicolons, and a closing parenthesis.
The syntax for each parameter declaration is

parameter declaration
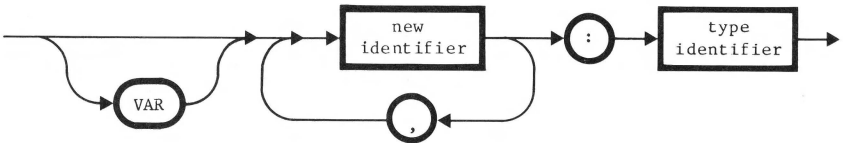
If the word VAR is used, this declaration declares one or more
variable parameters; otherwise it declares one or more value
parameters. The distinction is explained below. Each
declaration can declare any number of parameters, all of the same
type. Note that the type must be given as a single identifier
such as REAL, CHAR, or the identifier of type that has been
declared in the program. This is one of the reasons for
declaring types.

Here is an example of a simple procedure heading that declares
three value parameters:

        PROCEDURE ALPHA (INITIAL, LIMIT: REAL; COUNT: INTEGER);

INITIAL and LIMIT are real parameters, and COUNT is an integer
parameter. The following example declares a variable parameter
and a value parameter:

        PROCEDURE BETA (VAR ERRFLAG: BOOLEAN; N: INTEGER);

ERRFLAG is a variable parameter of type boolean, and N is a value
parameter of type integer.

The rest of a procedure definition consists of one block. The
syntax for a block is

block
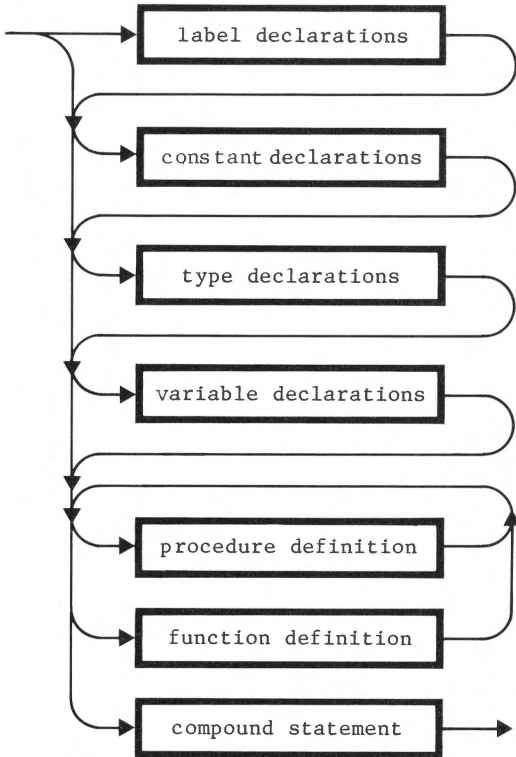
In other words, a block consists of optional declarations, optional procedure and function definitions, and one compound statement. Further on in this chapter we will see a special case where the block is replaced by the word FORWARD.

Value Parameters

There are two kinds of parameters: value parameters and variable parameters. Every parameter is a value parameter unless it is explicitly declared as a variable parameter (see next section). Value parameters are used to pass values (of expressions or variables) to a procedure or function at the time it is called.

The following example shows how value parameters can be used:

```
PROCEDURE WRITEMEAN (A, B: REAL);
  {Display the mean of two real values}
  VAR SUM: REAL;
  BEGIN
    SUM := A + B;
    WRITELN(SUM/2) {Display the value on the screen}
  END;
```

WRITEMEAN has two <u>formal parameters</u>, A and B; both are value parameters of type real. A and B are, in effect, real variables belonging to the WRITEMEAN procedure; but they have the special property that each time the procedure is called, A and B are initialized with the values of <u>actual parameters</u> contained in the call statement. The call statement must provide an actual parameter for each formal parameter.

The actual parameters are expressions. Each expression is evaluated and the result is assigned to the corresponding formal parameter before the statements of WRITEMEAN are executed.

All of the following are valid calls to WRITEMEAN (with different results):

```
WRITEMEAN(4.3, X);
WRITEMEAN(X, Y);
WRITEMEAN(Z + 2.3*Y, X);
WRITEMEAN(25, Z)
```

In these procedure calls, assume that X, Y, and Z are variables or constants of type real. Note that each actual parameter in the procedure call is an expression; the expression is evaluated and the value is passed to the procedure. The fourth example shows that an integer value may be supplied for a real parameter; the integer value is converted to a real value just as if it were being assigned to a real variable.

The type of a value parameter can be any Pascal data type except a file type. To pass a file to a procedure or function, you must use a variable parameter (see next section).

Variable Parameters

A value parameter, as we have seen, provides one-way
communication between the calling program and the procedure or
function: the call supplies a value, and this value is used
inside the procedure or function. A variable parameter provides
two-way communication.

With a variable parameter, the actual parameter is not an
expression but a variable reference, and the information passed
to the procedure or function is not the value of the variable but
the variable itself. Note that this variable is one declared
outside the procedure or function.

(At run time, what is passed is the address of the actual
parameter, so that the code of the procedure or function can
access it.)

If the value of the formal parameter is changed inside the
procedure or function, the effect is to change the value of the
actual parameter variable (outside the procedure or function).
The declaration of the formal parameter is preceded by the
reserved word VAR, as in the following example:

```
PROCEDURE MOVIT (RHO, THETA: REAL; VAR X, Y: REAL);
   {Update rectangular coordinates X and Y for motion
    through distance RHO at angle THETA (in degrees).}
   CONST PI = 3.14159;
   BEGIN
     THETA := THETA*PI/18Ø;           {Convert to radians}
     X := X + (RHO * COS(THETA));
     Y := Y + (RHO * SIN(THETA))
   END;
```

COS and SIN are trig functions contained in the system library
(see Appendix A). They assume that angles are given in radians.
The MOVIT procedure has two value parameters, RHO and THETA, and
two variable parameters, X and Y. Suppose that MOVIT is called
by the statement

    MOVIT(RADIUS, ANGLE, HORIZ, VERT)

where RADIUS, ANGLE, HORIZ, and VERT are real variables. MOVIT

assigns the current values of RADIUS and ANGLE to its own value
parameters RHO and THETA, respectively. It also assigns the
current values of HORIZ and VERT to X and Y respectively.

When MOVIT executes, it converts the value of THETA to radians
without affecting the variable ANGLE. But when it changes the
values of X and Y, it changes the values of HORIZ and VERT.
Because MOVIT was called with HORIZ and VERT as actual parameters
for the formal variable parameters X and Y, each reference to X
during this execution of MOVIT is in effect a reference to HORIZ;
and each reference to Y is in effect a reference to VERT.

Of course, MOVIT could have been written with direct reference to
HORIZ and VERT, instead of X and Y. But that would make it less
flexible; by using variable parameters, MOVIT is able to update
any two real variables that are passed to it.

The type of a variable parameter can be any data type, including
file types. If you have a formal parameter that is a string, you
can pass any string variable to it, even if the declared length
is not the same. However, if the length of the source parameter
is less than the length of the formal parameter, you must use the
VARSTRING Compiler option, which is described in Appendix F.

However, an individual element of a packed variable cannot
be supplied as the actual parameter (see Chapters 7 and 8
for discussion of packed variables).

For special purposes, there are two data types called
BYTESTREAM and WORDSTREAM which are used for variable
parameters. Chapter 13 describes these types.

Defining a Function

Syntactically, a function definition is very similar to a
procedure definition. The heading has the word FUNCTION instead
of PROCEDURE, and it specifies a type, which is the type of the
value returned by the function.

function definition
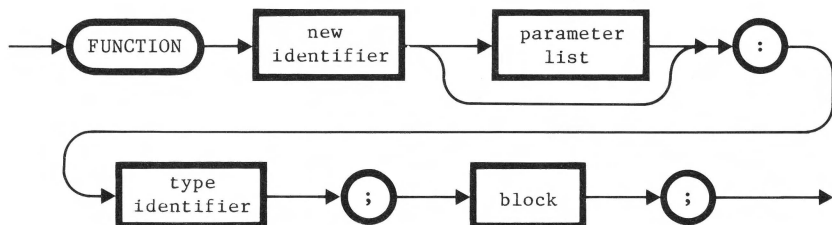
The type of the function must be a simple type (real, scalar, subrange, or pointer).

Within the function, there should be an assignment statement that has the function identifier on the left-hand side. This is how the function returns a value. For example, consider the WRITEMEAN procedure shown earlier. It displays a result on the screen. For some programs, it would be more useful to define a function which would perform the same calculation and return the result:

```
FUNCTION MEAN (A, B: REAL): REAL;
   {Return the mean of two real values}
   VAR SUM: REAL;
   BEGIN
     SUM := A + B;
     MEAN := SUM/2
   END;
```

If no value is assigned to the function identifier, an undefined value will be returned. If there are two or more assignment statements with the function identifier on the left-hand side, the last value assigned as the function executes is the value returned.

Normally the function identifier should be used within the function only as shown in the example above: on the left side of an assignment statement, for the purpose of returning a value. Do not use the function identifier on the right-hand side of an assignment statement within the function, unless the function is designed to be recursive. Recursive functions and procedures are discussed further on.

Calling a Function

As mentioned in Chapter 2, a function is activated by a function
call, which appears as an operand in an expression. When the
operand is evaluated at run time, the function is executed.   The
value returned by the function becomes the value of the operand.
The syntax of a function call is

function call
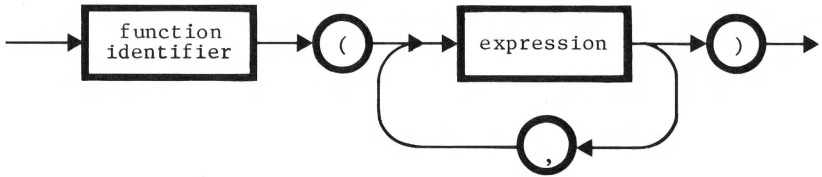
Recursion

A recursive procedure or function is one that calls itself; this
is permitted in all Pascal procedures and functions.   A full
discussion of the idea of recursion is beyond the scope of this
manual; instead a single example is offered as an illustration.

Consider the following situation: A 3Øx5Ø array of boolean
values is used to represent a graphic picture 3Ø dots high and 5Ø
dots wide.   Each boolean value represents a dot in the
picture--TRUE for a white dot and FALSE for a black dot.   The
array is declared as follows:

        VAR PIC: ARRAY[1..3Ø, 1..5Ø] OF BOOLEAN;

Now suppose that the picture contains several "images," each of
which consists of a group of white dots that are connected to
each other--that is, every dot in an image is a neighbor of at

least one other dot in the image (horizontally, vertically, or diagonally). In terms of the array, this means that an image is a collection of TRUE elements, and each of these elements is a neighbor of at least one other element in the image. One array element is a neighbor of another if their indices differ by 1 or $\emptyset$, and both have the value TRUE.

The problem is this: we want to write a procedure called ZAP that will erase all the dots in one image, if we give it the coordinates (indices) of any one dot in the image—without affecting any other image in the picture. This is an unwieldy problem if ZAP has to find all the dots in the image in a single pass; but if ZAP can call itself, the problem becomes simple. In English, the ZAP procedure is

"IF the specified dot is in the array AND it is white,
THEN erase it and ZAP each of its neighbors in turn.
ELSE do nothing and return immediately."

When ZAP is called once with the indices of a dot in an image, it will eventually call itself for every other dot in the image. To write ZAP in Pascal, we first write a convenient function for checking that a pair of indices is valid—i.e., that both indices are within the bounds of the array PIC:

```
FUNCTION INARRAY(I, J: INTEGER): BOOLEAN;
   BEGIN
      INARRAY := (I IN [1..3∅])
                    AND
                 (J IN [1..5∅])
   END;
```

The reference INARRAY(A, B), where A and B are integer values, will return TRUE if A and B are both valid indices for PIC. Now we can write ZAP as follows:

```
PROCEDURE ZAP(X,Y: INTEGER);
{Two variables to be used as coordinates of neighbors: }
VAR XN, YN: INTEGER;
BEGIN
  {If X,Y is in the array and is a white dot... }
  IF INARRAY(X,Y) THEN IF PIC[X,Y] THEN BEGIN
  {...then erase it... }
  PIC[X,Y] := FALSE;
  {...and ZAP all its neighbors: }
  FOR XN := X-1 TO X+1 DO
    FOR YN := Y-1 TO Y+1 DO
      ZAP(XN,YN)
  END
END;
```

(Notice that in the process of ZAPping all the neighbors, ZAP will also ZAP the dot that it started with. This is harmless, because the dot is no longer white; this particular recursive call will do nothing and return immediately.)

Each time a recursive routine calls itself, a new "incarnation" of the routine is created--that is, all the data belonging to the current incarnation has to be saved and new space allocated for the data belonging to the new incarnation. Eventually, if the routine is written correctly, the recursion terminates; the last incarnation does not call itself, but simply returns to the previous incarnation, which returns to the one before it, and so forth. Finally the first incarnation returns, and the execution of the recursive routine is finished.

In the case of ZAP, each incarnation makes 9 recursive calls in sequence. Each of these calls starts a new chain of incarnations. Each chain terminates when the dot that it is supposed to ZAP turns out not to be in the array, or not to be a white dot. When all the dots in the image have been erased, then all the chains have terminated and all the incarnations have returned; the original incarnation returns to the point in the program where ZAP was called non-recursively.

Termination

In order to make sense, a recursive function or procedure has to be written so that it will always terminate. This means that there is some condition under which it will not call itself; furthermore, if it calls itself enough times, it will always arrive at that condition. Otherwise, it may keep calling itself

until the system runs out of space to keep track of all the recursive calls, and halts the program with a "Stack overflow" error message. Actually this can happen even if the recursive procedure or function is correctly written, because of the space required by the numerous incarnations. When this happens, some sort of rewriting is required. A full discussion of space-saving techniques is beyond the scope of this manual, but the following suggestions may prove helpful:

- Eliminate the recursion, or find a way to make it terminate sooner.

- Reduce the amount of storage used for variables inside the recursive routine, since this storage has to be replicated for each recursive call.

- Segment your program to increase the amount of space available at the point where the recursive routine is called (see Chapters 14 and 15).

- If the program uses dynamic variables, use the MARK and RELEASE procedures to increase the amount of space available at the point where the recursive routine is called (see Chapter 9).

- Use files to store large data structures on diskette instead of in memory (see Chapters 1∅, 11, and 12).

Indirect Recursion

The above discussion describes direct recursion; there is also such a thing as indirect recursion. Suppose that a program contains three or more procedures or functions called A, B, C, and so forth. If A calls B and B calls A, that is indirect recursion. Likewise, if A calls B, B calls C, and C calls A, we have indirect recursion. The most general definition of recursion is that it occurs whenever a procedure or function is called (by itself or by another procedure or function) before it completes its execution and returns.

Like direct recursion, indirect recursion requires that there be a condition that will terminate the recursion, and the procedures must be designed to guarantee that the termination condition will be reached.

When you write indirectly recursive procedures or functions, one
procedure or function must call another before it is defined.
This is impossible using normal procedure or function
definitions, so Pascal provides a special form of definition
called the forward definition. In a forward definition the block
is replaced by the word FORWARD. The forward definition suffices
to declare the identifier and parameters, and the type in the
case of a function definition. The forward-defined procedure or
function can then be called in a following procedure or function,
and then the remainder of the forward definition can be given as
in the following example (where function F calls procedure P and
vice versa):

```
{Forward definition of F, to allow it to be referenced
within P:}
FUNCTION F (X, Y: REAL; COUNT: INTEGER): REAL;
   FORWARD;

{Normal definition of P, which calls F:}
PROCEDURE P (N: INTEGER);
   VAR A, B, C: REAL;
   BEGIN
      ...                      {Various statements}
      C := 2 *  F(A, B, N)   {This calls F}
      ...                      {Various statements}
   END;

{Continued definition of F; parameters and type omitted
since they are already declared:}
FUNCTION F;
   VAR TMP, DL, DX, DY: REAL;
   BEGIN
      ...                      {Various statements}
      P(TRUNC(X))            {This calls P}
      ...                      {Various statements}
   END;
```

Rules of Scope

Up to this point we have informally talked about a variable
"belonging to" a particular procedure or function, or to the main
program. What this means is that the procedure, function, or
program knows what the identifier means because it contains the
declaration.

Now we will give the formal rules for the <u>scope</u> of any object
that has an identifier. The scope of an object is that part of
the total program in which the object is known by its
identifier. The rules of scope are simple, and they apply
universally to declared constants, declared types, variables,
procedures, and functions.

In this section the word "procedure" will be used loosely, to
mean any procedure, any function, or the main program. Also the
word "declaration" will be used to include procedure and function
definitions. We can then view any program as a structure of
nested procedures. The main program itself is the outermost
procedure, and may have procedures nesting within it; these
nested procedures may have other procedures nested within them.
The "extent" of a procedure is the entire procedure, including
the heading and any procedures nested within it.

The rules of scope are:

- An identifier that has been used in a declaration in a
  particular procedure can be redeclared in any other
  procedure, including procedures nested within the first
  procedure.

- The scope of a declared object is the entire extent of
  the procedure in which it is declared, minus the entire
  extent of any nested procedure in which the same
  identifier is redeclared.

- The above rules apply to formal parameters, just as
  they apply to other variables declared in a procedure.

To see how this works, consider the program structure shown
below, where Procedures A and B are nested within Program P, and
Procedure Z is nested within Procedure B:

```
PROGRAM P;
   VAR FOO:  REAL;
       BAR:  INTEGER;

      PROCEDURE A;
         VAR FOO:  REAL;
         BEGIN
            ... {Statements of Procedure A}
         END;


      PROCEDURE B;
         VAR FOO:  BOOLEAN;

            PROCEDURE Z;
               VAR FOO:  INTEGER;
                   BAR:  CHAR;
               BEGIN
                  ... {Statements of Procedure Z}
               END;

         BEGIN
            ... {Statements of Procedure B}
         END;

   BEGIN
      ... {Statements of Program P}
   END.
```
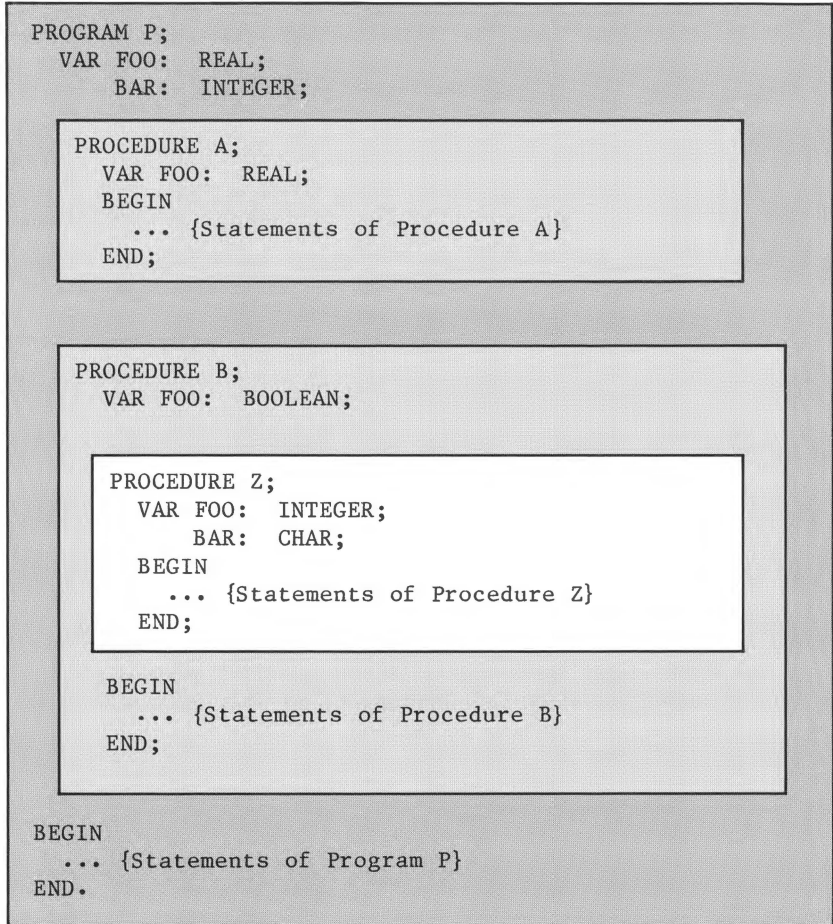
The identifiers FOO and BAR are declared and redeclared at
various points in the program. What is the scope of each
variable shown in the diagram?

   - The real variable FOO declared in the main program is
     known throughout the main program, except that it is
     not known anywhere within Procedure A or Procedure B

since the identifier FOO is redeclared in those
procedures.

- The integer variable BAR declared in the main program
  is known throughout the main program, except that it is
  not known anywhere within Procedure Z since the
  identifier BAR is redeclared in that procedure.

- The real variable FOO declared in Procedure A is known
  throughout Procedure A.

- The boolean variable FOO declared in Procedure B is
  known throughout Procedure B, except that it is not
  known anywhere within Procedure Z since the identifier
  FOO is redeclared in that procedure.

- The integer variable FOO and the char variable BAR
  declared in Procedure Z are known throughout Procedure
  Z.

A declared object is said to be local to the procedure in which
it is declared, and global to any nested procedure that is within
its scope. Thus each procedure knows about its own local
objects, and also about global objects. The nested structure of
Pascal, and the rules of scope, are what make it a block-
structured language. The virtues of block structure, and the
techniques for taking advantage of it, are beyond the scope of
this manual; however, any good tutorial on Pascal goes into this
topic at some length.

Built-in objects (the built-in types, procedures, functions,
etc.) act as if they were declared in a procedure that the main
program is nested in. Thus they are global to the main program,
unless they are redeclared. Values of variables local to a
procedure are lost upon exit from the procedure.

Segment Procedures and Functions

A segment procedure or function is declared by placing the word
SEGMENT at the beginning of the heading. For example,

    SEGMENT FUNCTION CALCULATE (X, Y, Z: REAL): REAL;

is a function heading for a segment function named CALCULATE.
The rest of the definition is conventional. The effect of making
the function a segment is that at run time, the code of the
function is not loaded into memory until the function is called;
as soon as it terminates, the space occupied by the code is
released and can be used for something else—such as the code of
another segment function or procedure. This is helpful with
programs that contain large procedures or functions; see Chapter
15 for more information.

Segment procedures and functions are called in the same way as
ordinary procedures and functions.

External Procedures and Functions

An external procedure or function is written in assembly language
as a .PROC or .FUNC. The assembled code is assumed to be in a
library file, which will be linked into the compiled Pascal
program before execution; the Program Preparation Tools manual
describes the use of the Assembler and Linker. Within the Pascal
program, the external procedure or function must still be defined
so that it can be called. The external procedure or function
definition consists of a conventional heading, with the word
EXTERNAL instead of a block. For example,

        PROCEDURE MAKESCREEN (INDEX: INTEGER);
          EXTERNAL;

This means that the procedure MAKESCREEN is an external
assembly-language procedure, with one parameter of type integer.
It is the user's responsibility to make sure that the assembly-
language procedure or function is compatible with the external
declaration in the Pascal program; the Linker checks only that
the parameters occupy the correct number of bytes.

There is a special rule for external procedures and
functions: a variable parameter can be declared without
any type.

Size and Complexity Limits

The Compiler imposes limits on the size and complexity of procedures. Here the term "procedure" includes functions and the main program. The "size" is the number of bytes of memory required for the compiled code of the procedure; "complexity" has to do with the number of backward jumps in the compiled code.

The Compiler error message "Procedure too long" means either that the procedure's code exceeds the limit of about 12∅∅ bytes, or that the procedure has too much complexity. In either case, the remedy is simple: move some statements from the offending procedure into one or more new procedures, and call the new procedure(s) at the point where the statements originally appeared. The new procedure(s) may be nested within the original procedure, so that the essential structure of the procedure is not changed.

By examining the source text of a procedure you cannot tell whether it will violate the limits, since the limits apply to the code created by the Compiler, not to the source text. But as a rule of thumb, a procedure whose body can be printed on one page will compile successfully. (Nested procedures don't count.) In any case it is good programming practice to keep procedures short; it makes the program much easier to understand and maintain.

7

Arrays, Sets, and Strings

Introduction

Up to this point, all the data types discussed have been simple
data types, which have single values. Pascal also has a variety
of structured data types, which can be thought of as collections
of values. This chapter covers only the three simplest kinds:
arrays, sets, and strings. Subsequent chapters cover records and
files.

An array variable is a collection of variables called elements;
all the elements of an array are of the same type. A single
element of the array is referenced by using the array identifier
with an index value (sometimes called a subscript). The index
value selects the desired element from among the other elements
of the array.

A set is a collection of values which are called members of the
set. Set operations in Pascal are fast, and allow very
straightforward coding of routines which would be much more
complicated without the use of sets.

A string is a sequence of characters, normally treated as a
single entity. Apple III Pascal provides a set of built-in
procedures and functions for manipulating strings.

Array Variables

An array variable is an ordered collection of elements, all of
the same type. Each element in an array can be considered as a
variable in its own right. A particular element is distinguished
from other elements by means of an index value enclosed in square
brackets. For example, you can declare an array called XYZ,
consisting of three elements of type real numbered 1, 2, and 3,
as follows:

        VAR XYZ: ARRAY [1..3] OF REAL;

Then these elements can be referred to individually as XYZ[1],
XYZ[2], and XYZ[3]. Each of them is a variable of type real.

Pascal arrays differ from arrays in other languages in several ways:

- Pascal arrays can have any number of dimensions.

- The elements of a Pascal array can be of any type except a file type. In particular, the elements can be arrays or records.

- The values used to index elements of a Pascal array can be of any scalar or subrange type except integer. (They can be a subrange of integer.) This means that the first element of an array is not necessarily element 0, or element 1; it depends on how the array is declared.

An array can be treated as a unit, without indexing, in three ways:

- It can be passed to a procedure or function as an actual parameter, if its type is "congruent" to the type of the formal parameter (this term is explained in Congruent Array Types later in this chapter).

- It can be assigned to another array of congruent type.

- It can be compared to another array of congruent type.

The syntax of an array type is:

array type
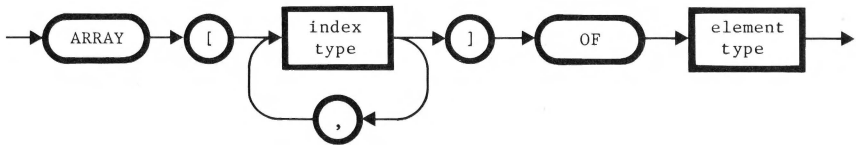
Note that there can be more than one index type: one for each dimension of the array type. First we will consider one-dimensional arrays.

One-Dimensional Arrays

An array type can be used in a type declaration as follows:

    TYPE VALS = ARRAY [0..99] OF REAL;

or in a variable declaration as follows:

    VAR VALUES: ARRAY [0..99] OF REAL;

The index type is usually a subrange of type integer, but it can also be any scalar or subrange type except the integer type. The element type can be any type except a file type.

The index type determines the number of elements in the array: there is one element for each possible value of the index. For example, consider the following declaration:

    VAR TENREALS: ARRAY [0..9] OF REAL;

The index type is the subrange 0..9, so the array TENREALS will have 10 elements, each one of which is of type real. The first element is TENREALS[0], the next is TENREALS[1], etc.; the last element is TENREALS[9].

The reason that the type integer is not allowed for array indices is simply that the array would have more elements than could be stored in memory.

Multidimensional Arrays

Since the type of the elements can be anything except a file type, you can declare an array of arrays. For example, here is a declaration of an array whose elements are arrays like the one declared above:

    VAR SQUARE: ARRAY [0..9] OF ARRAY [0..9] OF REAL;

SQUARE has ten elements, each of which is an array of ten real values. The variable SQUARE[3] is an array variable, and you can think of SQUARE as a 10x10 matrix of real values; SQUARE[3] can be thought of as a row, and its real elements can be thought of as the column elements of the row. To select one of the real values from the matrix, you need two indices; for example, SQUARE[6][5] refers to row 6, column 5.

Thinking of the first index as a row index and the second as a
column index is a matter of choice; you could just as well think
of the first index as a column index and the second as a row
index.

Instead of writing SQUARE[6][5], you can write both indices in
one pair of brackets, with a comma to separate them:
SQUARE[6,5]. The two notations are equivalent and
interchangeable. Similarly, you can condense the declaration of
SQUARE by writing

VAR SQUARE: ARRAY [Ø..9, Ø..9] OF REAL;

This declaration means exactly the same thing as the previous
one, and has the advantage of being more explicit to the human
reader. It obviously declares a two-dimensional array of reals.
Here is a declaration of a three-dimensional array:

VAR SPACE: ARRAY [Ø..MAXX, Ø..MAXY, Ø..MAXZ] OF REAL;

where MAXX, MAXY, and MAXZ are previously declared integer
constants. A Pascal array can have as many dimensions as
desired.

Other Index Types

So far, all the examples have shown integer subranges as index
types, since this is the most common usage. However, remember
that an index type can be any scalar type except integer. For
example, it can be char, or a subrange of char. The declaration

VAR CRYPT: ARRAY [CHAR] OF CHAR;

creates an array of characters which is indexed by character
values. It has one element for each possible character value, or
256 elements in all. Such an array could be useful for a
cryptography routine.

The indices of a multidimensional array can be of different
types. For a more complicated cryptographic scheme, you might
declare

VAR CRYPTARR: ARRAY [CHAR, 1..KEYMAX] OF CHAR;

where KEYMAX is a previously declared constant. CRYPTARR
contains one element for each possible combination of a character
value and an integer value from 1 to KEYMAX.

Index Values

In a reference to a specific element of an array, each index
value is given as an expression. For example, the following is a
valid assignment statement:

    SPACE[X,Y,Z] := SPACE[X-DX, Y-DY, K];

The only restriction on an expression used as an index value is
that the type of the expression's value must be compatible with
the index type in the array declaration; if the index type is a
subrange, the index value must be within the subrange.

Congruent Array Types

In the following sections the term "congruent type" is used. To
say that two array types are congruent means that they have
elements of the same type, the same dimensionality, and the same
number of elements in each dimension. However, the index type in
a particular dimension does not have to be identical for the two
arrays, as long as the number of elements in the dimension is the
same. For example, these three array types are congruent even
though they are not identical:

    TYPE A = ARRAY [∅..25, ∅..18] OF INTEGER;
         C = ARRAY [1∅..35, 1∅..28] OF INTEGER;
         D = ARRAY ['A'..'Z', ∅..18] OF INTEGER;

Each of these types is a 26-by-19 array of integers, and so they
are congruent types even though the index types are different.
The two array types

    TYPE E = ARRAY [1..5, 1..1∅] OF REAL;
         F = ARRAY [1..5∅] OF REAL;

are not congruent, even though both contain 5∅ real elements.
Type E is a two-dimensional 5-by-1∅ array and type F is one-
dimensional.

Passing Arrays

A procedure or function parameter (either value or variable) can be declared to be an array; then when the procedure or function is called, an array of congruent type can be passed as the actual parameter.

The following example shows a way to use this technique:

```
PROGRAM X;

CONST MINX = -1ØØ; MINY = -32; MINZ = Ø;
      MAXX = 1ØØ; MAXY = 156; MAXZ = 96;

TYPE XINDEX = MINX..MAXX;
     YINDEX = MINY..MAXY;
     ZINDEX = MINZ..MAXZ;
     THREESPACE = ARRAY [XINDEX, YINDEX, ZINDEX] OF REAL;

VAR MAINSPACE, BUFFERSPACE, SCRATCHSPACE: THREESPACE;
    NOMINAL, STRENGTH: REAL;
    ...

PROCEDURE INITSPACE (VAR S: THREESPACE; VALUE: REAL);
  {Set all elements of specified array to specified
   value.}
  VAR X: XINDEX; Y: YINDEX; Z: ZINDEX;
  BEGIN
    FOR X := MINX TO MAXX DO
      FOR Y := MINY TO MAXY DO
        FOR Z := MINZ TO MAXZ DO S[X,Y,Z] := VALUE
  END;
  ...

FUNCTION MEAN (A: THREESPACE): REAL;
  {Return the mean of all elements in array.}
  VAR X: XINDEX; Y: YINDEX; Z: ZINDEX; SUM: REAL;
  BEGIN
    SUM := Ø.Ø;
    FOR X := MINX TO MAXX DO
      FOR Y := MINY TO MAXY DO
        FOR Z := MINZ TO MAXZ DO SUM := SUM + A[X,Y,Z];
    MEAN := SUM / ( (MAXX-MINX+1)*(MAXY-MINY+1)*
                                  (MAXZ-MINZ+1) )
  END;
```

...

```
BEGIN
   ...
   INITSPACE(MAINSPACE, NOMINAL);
   ...
   STRENGTH := MEAN(MAINSPACE);
   ...
END.
```

The INITSPACE procedure has a variable parameter, S, which is an array of type THREESPACE. In the main program, INITSPACE is called with the array MAINSPACE as actual parameter; this is legal because MAINSPACE is also of type THREESPACE. When INITSPACE assigns the value of NOMINAL to each element of S, the effect is to assign the value of NOMINAL to each element of MAINSPACE.

The MEAN function has a value parameter, A, which is an array of type THREESPACE. It returns the mean of the values of the elements of the array. In the main program, MEAN is called with MAINSPACE as its actual parameter; it therefore returns the mean of all the element values of MAINSPACE.

Array Assignments

An array can be assigned to another array of congruent type. For example, if we have the declarations

```
VAR AAA, BBB: ARRAY [1..255] OF INTEGER;
```

then we can assign all the values of BBB to the corresponding elements of AAA as follows:

```
AAA := BBB
```

Array Comparisons

An array can be compared with another array of congruent type. The only comparison operators allowed for arrays are the = and <> operators (except as noted below under "Packed Character Arrays"). For example, if we have the declarations

```
VAR CCC, DDD: ARRAY [0..99] OF REAL;
```

then we can compare CCC to DDD as follows:

    CCC = DDD

The result of this expression is TRUE if every element of CCC has
the same value as the corresponding element of DDD. The other
possible comparison is

    CCC <> DDD

which is TRUE if any element of CCC has a different value from
the corresponding element of DDD.

Two packed arrays can be compared successfully only if all
16 bits in each word of each array have been defined. See
the following section.

Packed Arrays

Any array can be declared as a packed array by inserting the word
PACKED in the declaration as shown below.

Ordinarily, every scalar value or variable in Apple III Pascal
occupies one 16-bit word (two 8-bit bytes) of memory. This
includes scalar types such as boolean values, which can logically
be represented by a single bit, or char values which can
logically be represented in only eight bits. Obviously the use
of a whole word to store such a value is a waste of memory, but
not a significant waste in the case of single values.

If you have a large array with elements that can logically be
represented in less than one word, the waste of memory becomes
more significant. The most obvious example is large arrays of
characters, where the waste is 50% if each char value occupies a
word. The declaration

    VAR CHBUF: ARRAY [Ø..2Ø47] OF CHAR;

creates an array that occupies 2Ø48 words of memory, or 4Ø96
bytes. To avoid the waste, you can instead declare

    VAR CHBUF: PACKED ARRAY [Ø..2Ø47] OF CHAR;

This causes the char elements of the array to be packed, two in
each word. The packed array occupies only 1024 words or 2048
bytes.

When the program accesses an element of a packed array, a value
must be either unpacked from the array or packed into it. This
is done automatically, but requires some extra time for
execution. Thus the saving in storage space is a tradeoff
against execution speed and extra code space.

As will be seen in the next chapter, record types can also be
packed. In fact, you can insert the word PACKED into the
definition of any structured type, including sets and files.
However it has no effect except with arrays and records.

There is one restriction on the use of packed values: an element
of a packed array or record cannot be passed to a procedure or
function as an actual variable parameter.

In a packed array (or record), 16-bit words are still the unit of
storage; the difference is that one word may contain more than
one value. Values are never packed across a word boundary; for
example, consider this array:

    VAR XXX: PACKED ARRAY [1..3] OF 0..127;

To store a value in the range 0..127 requires just 7 bits,
logically. Thus two such values can be stored in one 16-bit
word, with two bits left over. The array XXX occupies two words
of memory: one word contains the elements XXX[1] and XXX[2], and
the second word contains the element XXX[3].

It is easy to be mistaken about how a particular array
will be packed. When in doubt, you can use the built-in
procedure SIZEOF to find out the actual number of bytes
occupied by any data type as explained in Chapter 13. The
following points about packing should be kept in mind if
you are trying to solve a critical space problem.

Since the word is the basic unit of storage, the minimum size for
any packed array is one word. Consider the array type

    TYPE EIGHTBITS = PACKED ARRAY [0..7] OF BOOLEAN;

Logically, this array type only requires one byte, since each of its 8 elements only requires one bit. However a variable of type EIGHTBITS actually occupies one word or 16 bits, since that is the minimum. Now consider

    VAR BATS: PACKED ARRAY [0..3] OF EIGHTBITS;

or its exact equivalent

    VAR BATS: PACKED ARRAY [0..3, 0..7] OF BOOLEAN;

BATS consists of four packed arrays, each containing eight booleans. You might expect BATS to occupy a total of 32 bits, or two words; but as we have just seen, a packed array of eight booleans takes a whole word. Therefore BATS occupies _four_ words, not two. Furthermore, note that

    VAR FATS: PACKED ARRAY [0..7, 0..3] OF BOOLEAN;

occupies _eight_ words!

In general, elements are packed only if each element can be represented in eight bits or less--in other words, only if two or more elements can be packed into a word. As the previous examples show, if the "elements" under consideration are arrays, they always require one or more words and can never be packed.

This leads to the fact that the word PACKED, in an array declaration, has an effect only when it appears just before the last occurrence of the word ARRAY. In other words, only the last dimension of an array can actually be packed. For example, the following two declarations are _not_ equivalent:

    VAR E: PACKED ARRAY [0..9] OF ARRAY [0..3] OF CHAR;
    VAR F: PACKED ARRAY [0..9, 0..3] OF CHAR;

The array E is not packed, because the word PACKED is not just before the last occurrence of the word ARRAY. The array F is packed, however.

Packed Character Arrays

One-dimensional packed arrays of characters have some special properties that go beyond what is allowed with other array types.

A string constant can be assigned to a one-dimensional packed
array of characters, if the number of elements in the array is
exactly the same as the number of characters in the string
constant. (See next section for details of string constants.)

Two one-dimensional packed arrays of char that have the same
number of elements can be compared using the >, <, <=, and >=
operators. The comparison is done as if the arrays were being
put in "alphabetical order" according to the ordering of the
ASCII character set. For example, if arrays A and B are declared
as follows:

     VAR A,. B: PACKED ARRAY [1..3] OF CHAR;

and during program execution the following assignments are made:

     A := 'cat';
     B := 'dog'

Then A contains the characters "cat" and B contains the
characters "dog". A is "less than" B, and the following
expressions are valid:

     A = B   {result is FALSE}
     A <> B {result is TRUE}
     A > B   {result is FALSE}
     A < B   {result is TRUE}
     A >= B {result is FALSE}
     A <= B {result is TRUE}

The same comparisons can be made between a one-dimensional packed
array of characters and a string constant, if the number of
elements in the array is the same as the number of characters in
the string constant. The comparisons above could be written as

     A = 'dog'   {result is FALSE}
     A <> 'dog' {result is TRUE}
     A > 'dog'   {result is FALSE}
     A < 'dog'   {result is TRUE}
     A >= 'dog' {result is FALSE}
     A <= 'dog' {result is TRUE}

One-dimensional packed arrays of characters are also handled in a
special way by the built-in procedures WRITE and WRITELN as
explained in Chapter 11.

Sets

In Pascal, a set is a collection of distinct values, all of the
same scalar type. The values are called <u>members</u> of the set; the
type of the members is called the <u>base type</u> of the set. The
syntax of a set type is

<u>set type</u>

Here are some examples of set variable declarations:

```
VAR LETTERS, SPECIALCHARS, PRINTINGCHARS: SET OF CHAR;
    DIGITS: SET OF '∅'..'9';
    COLORS: SET OF (VIOLET, BLUE, GREEN, YELLOW,
                        ORANGE, RED);
```

A set type specifies all the possible members of a set of that
type. For example, a SET OF CHAR can contain any collection of
distinct char values. The term "distinct" here means that a
particular char value can only appear once in the set; for
example, the set DIGITS, declared above, either contains the
character '5' or it doesn't.

When the type given after the words SET OF is a subrange, the
base type of the set is the base type of the subrange. Thus in
the second example above (SET OF '∅'..'9'), the base type of the
set DIGITS is the char type; the set can contain only the
characters '∅' through '9'.

Set Values

To understand set values, it may be helpful to know that a set
value is represented internally as a bit pattern, with a bit for
each possible member of the set. Each of these bits indicates
whether that possible member is actually a member. The point of

this is that although a set is a collection of members, a <u>set value</u> is a single value.

To write a set value explicitly, you specify its members between square brackets. For example, suppose that the variable SPECIALCHARS has been declared as a SET OF CHAR (as shown above). Now consider the assignment statement

        SPECIALCHARS := ['.', ',', ';', ':', '(', ')']

After this assignment, SPECIALCHARS is the set containing the period, comma, semicolon, colon, and left and right parentheses. Internally, this value is represented by a bit pattern containing 256 bits, one for each possible char value. In this pattern, the particular bits corresponding to the characters that are members of the set are "on" and the other bits are "off."

When the members form a subrange, you can use the subrange notation, as in

        DIGITS := ['∅'..'9']

This makes DIGITS the set of all characters from '∅' through '9'. Alternatively, if you wanted DIGITS to contain only octal digits, you could write

        DIGITS := ['∅'..'7']

This makes DIGITS the set of all characters from '∅' through '7'. You can also use more than one subrange and mix subranges and individual members as in the following:

        LETTERS := ['A'..'Z', 'a'..'z']
        COLORS := [VIOLET..GREEN, ORANGE];

After these two assignments, LETTERS is the set of all capital letters and all lower-case letters, and COLORS is the set containing VIOLET, BLUE, GREEN, and ORANGE.

A set value written with square brackets in this manner is called a <u>set constructor</u>. The syntax for a set constructor is

set constructor

```
──▶──( [ )──┬──▶──┌──────────┐──▶──( .. )──▶──┌──────────┐──┬──▶──────▶──( ] )──▶──
            │     │expression│               │expression│  │
            │     └──────────┘               └──────────┘  │
            │              ┌──────( , )◀──────┐            │
            └──────────────┘                  └────────────┘
```
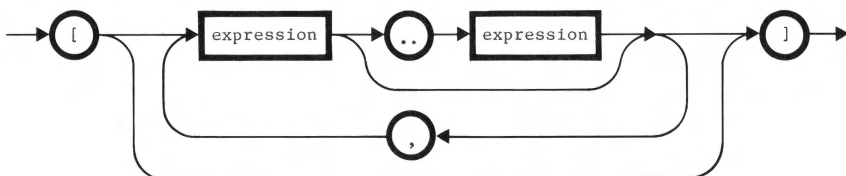
The set constructor [] denotes the empty set.

Restrictions on Sets

The base type of a set cannot have more than 512 values. A set
cannot contain any value whose ordinality is less than 0 or
greater than 511.   In particular, it cannot contain any integer
less than 0 or greater than 511.   An attempt to assign more than
512 members to a set, or to assign an integer member outside the
range $0..511$, results in a run-time error which halts the
program.

The formula for the number of words of storage, given the number
of members in a set, is

$$((n-1)\ DIV\ 16)+1$$

where n is the number of possible values in the base type.

A set of 512 members occupies 32 words of storage.

The IN Operator

The IN operator is used to test whether a particular scalar value
is a member of a particular set.   The IN operator is a relational
operator and has a boolean result.   It has the same precedence
as the other relational operators.

The IN operator must have a scalar value on the left and a set on
the right.   The type of the scalar value must be the same as the
base type of the set.   Suppose that we have the following
declarations and assignments:

```
TYPE COLOR = (MAGENTA, CYAN, YELLOW);

VAR LETTERS: SET OF CHAR;
    COLORS: SET OF COLOR;
    INCHAR, TSTCH: CHAR;
    INDEX: INTEGER;
    TINT: COLOR;

BEGIN
  LETTERS := ['A'..'Z', 'a'..'z'];
  COLORS := [MAGENTA, CYAN];
  ...
```

Then the following expressions are valid uses of IN:

```
INCHAR IN LETTERS {TRUE if value of INCHAR is a letter}
TSTCH IN ['a'..'z'] {TRUE if value of TSTCH is a lower-
                     case letter}
SUCC(TINT) IN COLORS {TRUE if successor of value of TINT
                      is CYAN}
(INDEX + 5) IN [0..255] {TRUE if value of (INDEX +5) is
                         in the range 0..255}
```

All of these expressions could be replaced with constructs that
do not use sets; for example, INCHAR IN LETTERS could be replaced
with

```
(INCHAR >= 'A') AND (INCHAR <= 'Z')
  OR (INCHAR >= 'a') AND (INCHAR <= 'z')
```

However, the expression INCHAR IN LETTERS is not only more
straightforward, it executes faster. Set operations in general
are quite fast.

Combining Sets

So far, we have only seen two ways to represent set values: set
variables, and set constructors. Such set values can also be
combined to form expressions with set results. The operators are
+, -, and *. These symbols are also used with numeric operands
to perform arithmetic. They have different meanings when the
operands are set values, but they have the same precedence
whether they function as set operators or arithmetic operators.

The + operator forms a set union. If A and B are set values with
members of the same type then the value of A + B (or B + A) is

the set that contains all members of A and all members of B.

For examples of the use of set unions, suppose that we have the declarations:

```
VAR CAPS, LOWERS, LETTERS, DIGITS, ALPHANUMERICS:
    SET OF CHAR;
```

and the assignments

```
CAPS := ['A'..'Z'];
LOWERS := ['a'..'z'];
DIGITS := ['Ø'..'9'];
```

Then we can conveniently make the following assignments for LETTERS and ALPHANUMERICS:

```
LETTERS := CAPS + LOWERS;
ALPHANUMERICS := LETTERS + DIGITS;
```

A common use of the union operator is to add a single new member to a set. Suppose that we wish to add the character '$' to the set ALPHANUMERICS:

```
ALPHANUMERICS := ALPHANUMERICS + ['$']
```

The — operator forms a set difference. If A and B are set values with members of the same type then the value of A — B is the set that contains all members of A that are not members of B. For example, the value of the expression

```
[CHR(Ø)..CHR(255)] - LETTERS
```

is the set of all character values that are not letters. You can also use the difference operator to remove a single value from a set. For example, the value of the expression

```
LETTERS - ['A']
```

is the set of all letters except the letter 'A'.

The * operator forms a set intersection. If A and B are set values with members of the same type then the value of A * B (or B * A) is the set that contains all members of A that are also members of B. For example, suppose that we have the declarations

```
VAR COMMANDS, OPTIONS: SET OF CHAR;
```

and the assignments

```
COMMANDS := ['A', 'S', 'M', 'D', 'E', 'O'];
OPTIONS := ['B', 'O', 'D', 'H']
```

then the value of the expression

```
COMMANDS * OPTIONS
```

is the set containing the characters 'O' and 'D'.

Comparing Sets

Two sets that have the same base type can be compared using the = and <> operators, to see if they are equal or unequal. They can also be compared using the <= and >= operators, to see if one set contains the other. These operators produce boolean results.

The same symbols are used for arithmetic comparison; they have different meanings when the operands are set values, but they have the same precedence whether they function as set comparisons or arithmetic comparisons. Note that the comparisons > and < cannot be used with sets.

With set operands, the = operator denotes set equality and the <> operator denotes set inequality. Two sets are said to be equal if they contain exactly the same elements, and unequal otherwise.

With set operands, the >= operator means "includes." Set A includes set B if every member of B is also a member of A. Note that A may contain other members which are not in B. Similarly, the <= operator means "is included in."

Strings

A string value is a sequence of up to 255 characters. Strings are supported by a set of built-in procedures and functions, described further on in this chapter.

String Constants

A string constant consists of the string itself between
apostrophes (with a special way of including apostrophes in the
string as explained below). The syntax is

string constant
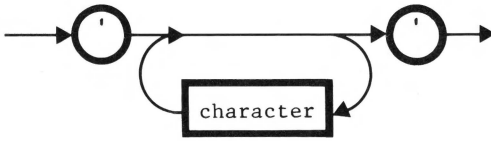
To write a string constant that contains an apostrophe, write two
apostrophes as in the following examples:

        'Can''t find the specified file.'
        'The challenger''s score is: '
        'Type either ''yes'' or ''no'''

A string constant can also be declared with an identifier, as in
the following:

        CONST ERRMSG = 'Can''t find the specified file';

Note that a string constant must be on a single line in the
program; it cannot contain a line-break.

String Variables

A string variable is a variable whose value at any point during
program execution is a string. String variables are usually
treated as single units, but it is possible to pick out a single
character value from a string variable by indexing it in the same
way that an array element is selected by indexing.

A string variable is created by declaring it with a string type.
The syntax of a string type is

string type
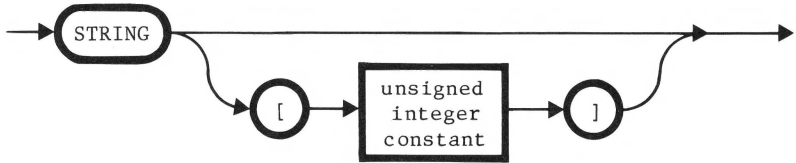
Here are some examples of string variable declarations:

```
VAR MSGBUFFER: STRING[255];
    INPUTNAME, OUTPUTNAME: STRING;
```

The number in brackets, if used, specifies the maximum length of the string. The number can be any integer from 1 through 255. If no number is specified, the maximum length is 8Ø. Since the string variable's value can change during execution, the system keeps track of the length of the string value; if this length exceeds the maximum, a run-time error occurs.

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 READLN procedure as described in Chapter 11:

```
READLN(TITLE)
```

or by means of the string built-ins, described further on in this chapter.

A string value can be compared to any other string value, regardless of length. Also, as previously mentioned, a string value can be compared to a one-dimensional packed array of characters if the length of the string is the same as the number of elements in the array.

The relational operators =, <>, <, >, <=, and >= are used. One
string is "less than" another if it would come first in an
"alphabetic" list of strings based on the ordering of the ASCII
character set.

Elements of a String

The individual characters within a string are indexed from 1 (not
Ø!) to the length of the string. For example, if TITLE is the
name of a string and we have the assignment

        TITLE := 'A Quick Brown Fox'

then TITLE[1] is a reference to the first character of TITLE,
namely the character 'A', and TITLE[17] is a reference to the
last character, namely 'x'. The index must not be less than 1 or
greater than the length of the string. For example, TITLE[18]
would lead to a run-time error.

There is a case where you must assign a character to a string
element. It may be necessary to convert the value of a char
variable to a one-character string value; for example you may
want to add it to a string with the CONCAT procedure (see next
section), which requires a string value rather than a char
value.

Note that while a one-character string constant is the same thing
as a character constant, a string variable whose value is a
single character is <u>not</u> the same thing as a character variable
and you cannot assign one to the other.

The way to deal with this is shown in the example below, which
assumes that the value of the char variable CHVAL is to be
concatenated to the value of the string variable LINE. To make
this possible we use a string variable named ONECH. First we
initialize it with a one-character string constant, so it will
have the right length, and then we assign the char value to its
element, ONECH[1].

        ONECH := 'X';
        ONECH[1] := CHVAL;
        LINE := CONCAT(LINE, ONECH)

Beware of zero-length strings: they cannot be indexed at
all without causing a run-time error. If a program

indexes a string that might have zero length, it should
first use the LENGTH function (see next section) to see if
the length is zero. If the length is zero, the program
should not execute statements that index the string.

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

String Built-Ins

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

The LENGTH Function

The LENGTH function returns the length of a string. LENGTH takes
one parameter:

    LENGTH (STRG)

where STRG is a string value parameter. For example, if we have
the assignment

    S := 'abcdefg';

then the value of LENGTH(S) is 7 and the value of S[LENGTH(S)]
is 'g'.

The POS Function

The POS function returns an integer value. POS takes two
parameters:

    POS (SUBSTRG, STRG)

where both SUBSTRG and STRG are string value parameters. 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.

For example, suppose that a string variable named FNAME contains a filename that has been typed on the keyboard. Then the value of POS('.', FNAME) will be ∅ if the filename contains no period, 1 if the first character is a period, etc.

The CONCAT Function

The CONCAT function returns a string value. CONCAT can take any practical number of actual parameters each of which is a string value; the parameters are separated by commas. The syntax of the call is
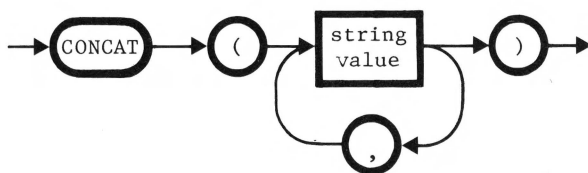
CONCAT returns a string which is the concatenation of all the strings passed to it. For example, if FNAME is a string variable containing a file name from the keyboard, then the statement

        FNAME := CONCAT(FNAME, '.TEXT')

has the effect of appending the suffix .TEXT to the name. Another example: if we have the assignments

        FIRSTNAME := 'George';
        LASTNAME := 'Washington'
        •••
        BOTHNAMES := CONCAT(LASTNAME,', ',FIRSTNAME)

then the statement

        WRITELN(BOTHNAMES)

will print

        Washington, George

The COPY Function

The COPY function returns a string value. COPY takes three
parameters:

    COPY (STRG, INDEX, COUNT)

where STRG is a string value parameter and both INDEX and COUNT
are integer value parameters. COPY 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.
DELETE takes three parameters:

    DELETE (STRG, INDEX, COUNT)

where STRG is a string variable parameter and both INDEX and
COUNT are integer value parameters. DELETE removes COUNT
characters from STRG starting at the INDEX specified. Example:

    LATIN := 'Cartago delenda est.';
    DELETE(LATIN, POS('delenda', LATIN), LENGTH('delenda '));
    WRITELN(LATIN)

This will print:

    Cartago est.

The INSERT Procedure

The INSERT procedure modifies the value of a string variable.
INSERT takes three parameters:

    INSERT (SUBSTRG, STRG, INDEX)

where SUBSTRG is a string value parameter, STRG is a string
variable parameter, and INDEX is an integer value parameter.
INSERT 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

You can insert a substring at the end of a string by using INDEX
value LENGTH(string)+1. For example

```
S := 'ABC'
T := 'DE'
INSERT(T,S,LENGTH(S)+1);
```

will produce a string of length 5 containing

    'ABCDE'.

The STR Procedure

The STR procedure modifies the value of a string variable. STR
takes two parameters:

    STR (N , STRG)

where N is an integer value parameter, and STRG is a string
variable parameter. N may be a long integer.

STR converts the value of N into a string. The resulting string
is placed in STRG. Example:

```
INTLONG := 102039503;
STR(INTLONG, INTSTRING);
INSERT('.', INTSTRING, LENGTH(INTSTRING)-1);
WRITELN('$', INTSTRING)
```

This will print:

$1020395.03

The use of STR requires availability of the long integer procedure, LONGINTIO, which is found in the standard system library file.

8

Records

Record Variables

A record variable is a collection of elements called <u>fields</u> which may be of different types. Each field has its own identifier within the record, and can be individually referenced; or the record can be referenced as a whole.

As will be seen in later chapters, record types are extremely useful in conjunction with dynamic variable allocation and files; in this chapter, the discussion is restricted to ordinary record variables (which are neither dynamic variables nor components of files).

The syntax for a record type is

<u>record type</u>

→(RECORD)→[field list]→(END)→

The syntax for a field list is

<u>field list</u>

→[new identifier]→(:)→[type]→(;)→[variant part]→
                 ↖(,)↙   ↖(;)↙
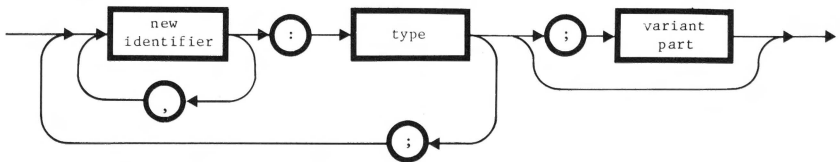
Each field identifier is the name of a distinct variable, or field, within that record type. The type of a field may be anything except a file type. The syntax for the variant part is given in the next section; in this section, we assume that there is no variant part.

The following record type can be used to represent a date:

```
DATE = RECORD
           DAY, YEAR: INTEGER;
           MONTH: STRING
       END;
```

If you declare a variable named TODAY, of type DATE, then the fields of this variable can be referenced as TODAY.DAY (an integer variable), TODAY.YEAR (another integer variable), and TODAY.MONTH (a string variable).

The record type below uses the type DATE for two of its fields. It could be used in a checking-account program.

```
TYPE CHECK = RECORD
                 CHECKNUMBER: INTEGER;
                 DATEWRITTEN, DATEPAID: DATE;
                 AMOUNT: REAL;
                 RECIPIENT: STRING;
                 BOUNCED: BOOLEAN
             END;
```

Type CHECK contains six fields. A checkbook might be represented as an array of records of type CHECK:

```
VAR CHECKBOOK: ARRAY[1..100] OF CHECK;
```

To reference a field in a record, merely write a reference to the record, then a period, then the field identifier. For example, with the declarations above,

- CHECKBOOK[3] refers to a particular check (a variable of type CHECK).

- CHECKBOOK[3].CHECKNUMBER refers to the number of that check (an integer variable).

- CHECKBOOK[3].DATEWRITTEN refers to the date on which that check was written (a variable of type DATE).

- CHECKBOOK[3].DATEWRITTEN.MONTH refers to the month within the date (a string variable).

To assign numbers to the checks in the array you could use the statement:

        FOR I := 1 TO 1ØØ DO CHECKBOOK[I].CHECKNUMBER := I;

Variant Records

The optional variant part of a record type contains two or more alternate field lists. For example, a variant part of a record could be declared to contain either a string or two reals, an integer, and a string.

To understand what this means, recall that a Pascal variable consists of two things:

  - Allocated memory space to hold the value of the variable in binary form

  - Type information which tells the system how to interpret the binary information.

When a variant part is declared, the Compiler allocates enough space for the largest of the alternate field lists in the variant part. All of the alternate field lists then use the same allocated space.

The syntax for a variant part is

variant part
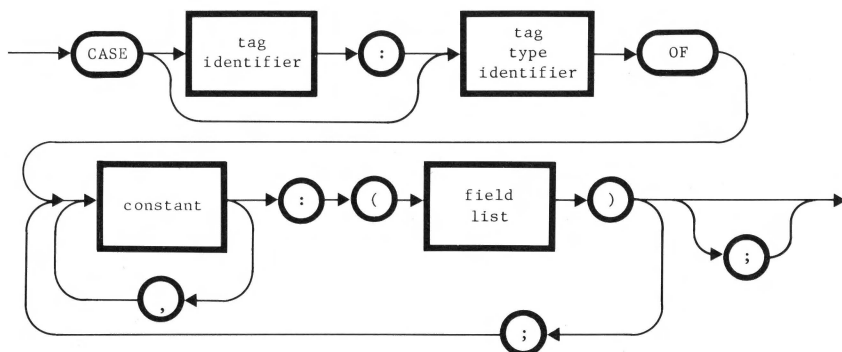
The variant part resembles a CASE statement, and its meaning is closely related. The tag identifier and tag type serve to declare a **tag field**, which can be of any scalar type. The tag field is an ordinary field of the record (just as if it were declared before the variant part).

Note that the tag identifier is optional; if it is omitted, then there is no tag field. In the rest of this chapter, we assume that the tag field is present; for information on the use of variants without tag fields, see Appendix G.

The tag type cannot be omitted, because it also relates to the constants within the variant part: each constant in the variant part must be one of the possible values of the tag identifier. Each of these constants is a "label" for a field list, enclosed in parentheses.

The variant part syntax allows you to declare records of a single type which can have more than one configuration -- each of the field lists in the variant part represents a specific, distinct configuration.

At run time, the program can refer to any of the fields in the variant part. All of the "cases" or field lists of a variant

part occupy the same space in memory, on the assumption that at any point in the program, the data in that space is to be interpreted according to just one field list.

The following example shows a record that has a variant part.

```
TYPE YESNO = (YES, NO);
     IDENT = RECORD
               LASTNAME, FIRSTNAME: STRING[1Ø];
               CASE HASLIC: YESNO OF
                 YES: (LICNO: INTEGER[1Ø]);
                 NO: (SOCSEC: STRING[1Ø])
             END;
```

A record of type IDENT contains four fields: LASTNAME, FIRSTNAME, HASLIC and either LICNO or SOCSEC. The program can use the tag field HASLIC to determine which variant field, LICNO or SOCSEC, should be used.

Fields in the variant part of a record are referenced exactly as normal fields are. At any point in the program, you can refer to either IDENT.LICNO or IDENT.SOCSEC. The two references refer to the same physical data; the difference is that a reference to IDENT.LICNO interprets the data as a ten-digit long integer, while a reference to IDENT.SOCSEC interprets the same data as a string of up to ten characters.

Now suppose we have an array of records of type IDENT, and some other variables to contain information input by the user:

```
VAR IDCARD: ARRAY [1..1ØØØ] OF IDENT;
    LASTNAMEIN, FIRSTNAMEIN, SSNUMIN: STRING[1Ø];
    HASLICIN: YESNO;
    LICNUMIN: INTEGER[1Ø];
```

The following assignments can be made to a particular record, IDCARD[N]:

```
IDCARD[N].LASTNAME := LASTNAMEIN;
IDCARD[N].FIRSTNAME := FIRSTNAMEIN;
IDCARD[N].HASLIC := HASLICIN;
CASE IDCARD[N].HASLIC OF
    YES: IDCARD[N].LICNO := LICNUMIN;
    NO:  IDCARD[N].SOCSEC := SSNUMIN
END
```

Notice that the tag field is used to select the proper variant field.

The tag field does not have to be declared after the word CASE; as mentioned before, it is really just another field of the record. In fact, the type IDENT declared above can be thought of as an abbreviation for the declaration

```
TYPE IDTOO = RECORD
             LASTNAME, FIRSTNAME: STRING[1Ø];
             HASLIC: YESNO;
             CASE YESNO OF
                YES: (LICNO: INTEGER[1Ø]);
                NO:  (SOCSEC: STRING[1Ø])
             END;
```

where the tag field is declared before the variant part. However, note that the tag type must appear after the word CASE, since it determines the possible values for the constants in the variant part.

The tag field does not automatically control which fields can be referenced in the variant part of the record. It is up to the program to use the tag field as a control, as shown in the example above.

The WITH Statement

The WITH statement is a shorthand method for referencing elements of a record. It provides a means by which the fields of specified records can be referenced using only their field identifiers. The syntax of a WITH statement is

with statement

WITH → record variable reference → DO → statement

,
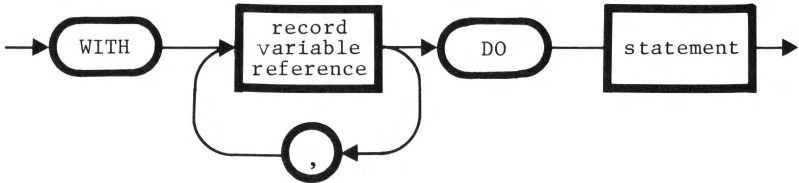
The meaning of the record variable reference is determined once, before the statement following DO is executed.

Earlier, we showed the following assignments to the record IDCARD[N]:

```
IDCARD[N].LASTNAME := LASTNAMEIN;
IDCARD[N].FIRSTNAME := FIRSTNAMEIN;
IDCARD[N].HASLIC := HASLICIN;
CASE IDCARD[N].HASLIC OF
   YES: IDCARD[N].LICNO := LICNUMIN;
   NO:  IDCARD[N].SOCSEC := SSNUMIN
END
```

These statements can be abbreviated by using a WITH statement:

```
WITH IDCARD[N] DO BEGIN
  LASTNAME := LASTNAMEIN;
  FIRSTNAME := FIRSTNAMEIN;
  HASLIC := HASLICIN;
  CASE HASLIC OF
     YES: LICNO := LICNUMIN;
     NO:  SOCSEC := SSNUMIN
  END
END
```

Variables that are not fields of records may be referenced as usual within the WITH statement. For example, if COUNTER is declared as a variable of type integer, then you can write

```
WITH CHECKBOOK[I] DO BEGIN
  COUNTER := COUNTER+1;
  ...
  CHECKNUMBER := COUNTER MOD 1000
END
```

The identifier COUNTER references the integer variable COUNTER
(rather than a record field named COUNTER) because the records
listed do not have a field named COUNTER.

If one of the fields of a record is itself a record, then nested
WITH statements can be used:

```
WITH CHECKBOOK[I] DO WITH DATEWRITTEN DO BEGIN
   CHECKNUMBER := I;
   DAY := 5;          {references CHECKBOOK[I].DATEWRITTEN.DAY}
   MONTH := 'JULY'
END
```

For convenience, these nested WITH statements can be combined
into a single WITH statement:

```
WITH CHECKBOOK[I], DATEWRITTEN DO BEGIN
   CHECKNUMBER := I;
   DAY := 5;          {references CHECKBOOK[I].DATEWRITTEN.DAY}
   MONTH := 'JULY'
END
```

When the same field name occurs in more than one record, and both
records are "abbreviated" via the WITH statement, a potential
ambiguity arises. Consider the following:

```
WITH CHECKBOOK[I], DATEWRITTEN, DATEPAID DO BEGIN
   •••
   DAY := 5;
   •••
END
```

The field DAY occurs both in CHECKBOOK[I].DATEPAID.DAY and in
CHECKBOOK[I].DATEWRITTEN.DAY; which one is referenced in the
assignment statement? The answer is that
CHECKBOOK[I].DATEPAID.DAY is referenced, because DATEPAID is the
last record listed in the WITH statement.

The confusion arises because DATEWRITTEN and DATEPAID are
"parallel;" that is, one is not a field of the other. WITH
statements that allow this kind of confusion should be avoided.
For example, the following WITH construction can be used to
reference fields of the same name in both DATEWRITTEN and
DATEPAID:

```
WITH CHECKBOOK[I] DO BEGIN
  WITH DATEWRITTEN DO BEGIN
    •••
    DAY := 5;
    •••
  END;
  WITH DATEPAID DO BEGIN
    •••
    DAY := 6;
    •••
  END
END
```

Comparisons and Assignments

Although most assignments to records are done on a field by field
basis, it is also possible to make assignments between entire
records of the same type. For example, the result of the
assignment

    CHECKBOOK[I] := CHECKBOOK[I + 1]

is to assign to every field in CHECKBOOK[I] the value of its
corresponding field in CHECKBOOK[I + 1].

Actually, assignments between records of different types
are allowed if the types are "congruent" -- that is, if
every field in one record has a corresponding field, of
the same type but not necessarily of the same name, in the
other record.

The only operations that can be performed on records are
comparisons, giving boolean results. Of the relational
operators, only

    =   equal to
    <>   not equal to

can be used with records. The rules for comparing two records
are the same as the rules for assignment to records: the two
records being compared must have corresponding fields of
identical types.

Two packed records can be compared successfully only if
all 16 bits in each word of each record have been
defined. See the following section.

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 packed.

```
VAR R: PACKED RECORD
         I,J,K: 0..31;
         B: BOOLEAN
       END;
```

The fields I, J, K each take up five bits in the word. The
boolean field B occupies the 16th bit of the same word.

In much the same manner that multidimensional arrays can be
packed, packed records may contain fields which themselves are
packed records or packed arrays. 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[0..3]             H: PACKED ARRAY[0..3]
            OF CHAR                           OF CHAR
       END;                              END;
```

As with packed arrays, the word PACKED should appear with every
occurrence of the word RECORD or ARRAY to ensure that all fields
of the record are actually packed. In the above example, only
record A has the F field 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 variant part may be used in a packed record, and the amount of space allocated to it will be the size of the largest variant among the various cases. The details of the packing methods are beyond the scope of this document.

```
VAR K: PACKED RECORD
        B: BOOLEAN;
        CASE F: BOOLEAN OF
           TRUE: (Z: INTEGER);
           FALSE: (M: PACKED ARRAY[0..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 variant part is always the size of the largest variant, so in the above example the variant part will occupy two words, one byte for each of four characters. Thus the entire packed record will occupy three words.

9

Pointers and Dynamic Variables

Concepts

Up to this point we have dealt only with static variables: these
are declared in the program and the Compiler responds to the
declaration by

- allocating memory space to hold the value of the
  declared variable at run time;

- associating the declared identifier with that memory
  space, and with the type information in the
  declaration.

In other words, everything about a static variable except its
actual value is determined when the program is compiled. Only
the value is determined at run time. Variant records may appear
to be an exception to this statement, but really they are just a
slightly more complicated case; the Compiler allocates enough
space for the largest variant of a record variable, and at run
time the field identifier determines how this space is used.
Again, all information about the variable is determined at
compile time, except for the actual value(s).

Some programs need a more flexible kind of variable. For
example, imagine a program that will read a sequence of records
from a file or from the terminal; suppose that the records are of
type

        TYPE RTYPE = RECORD
                        INTVALS: INTEGER;
                        RVAL: REAL;
                     END;

At the time when the program is written, the difficulty is that
we don't know how many records will be read. How can we create
variables to hold the records? One way is to make a generous
guess and declare an array. For example if we think the number
of records will not exceed 1ØØ, we could declare

        VAR REC: ARRAY [1..1ØØ] OF RTYPE;

This is not very satisfactory; if the number does exceed 1ØØ the
program will be unable to cope, and if the number is less than

1∅∅ we will have wasted space. The situation calls for the use
of <u>dynamic variables</u>.

A dynamic variable is not declared, has no identifier, and is
created at run time. Since it has no identifier, it can only be
referenced indirectly, by means of a <u>pointer</u>. In Pascal,
pointers and dynamic variables are two parts of a single
mechanism. Note that this is the only Pascal mechanism for
accessing the unallocated memory space that is available at run
time.

Instead of declaring the REC array as shown above, we declare

        VAR PTR: ^RTYPE;

This creates a static variable called PTR which is a pointer to a
dynamic variable of type RTYPE. At compilation time there is no
such variable, but in the program we can write

        NEW(PTR)

NEW is a built-in Pascal procedure. The call shown does two
things at run time:

  - It creates a new variable of type RTYPE, somewhere in
    unused memory space.

  - It sets the value of PTR so that it points to this new
    variable.

Now we can refer to the new dynamic variable by writing

        PTR^

which means "the thing pointed to by PTR." We can refer to the
fields within the variable by writing PTR^.INTVALS or PTR^.RVAL,
just as if "PTR^" were the identifier of the variable. However,
another NEW(PTR) call will create a second variable of type
RTYPE, and then PTR will point to this second variable. At this
point you may be wondering how to refer to the first dynamic
variable after the second is created, and this point will be
dealt with at length further on.

Pointer Values

Pointer values are generated by NEW and there is no way to write
a pointer value explicitly. Therefore you cannot declare a
pointer constant; however, there is a built-in pointer constant
named NIL, which can be a value of any pointer variable and which
points to nothing.

The value of a pointer variable can be changed in only two ways:
by giving the pointer variable as a parameter to NEW and by
assigning it the value of another pointer variable or NIL.

The only other operation allowed with pointers is comparison
using the = and <> operators, yielding a boolean result.
Pointers cannot be compared using the <, <=, >, and >= operators,
and they cannot be combined using any of the arithmetic
operators. Pointers are not a scalar type, and thus do not have
successors or predecessors. However, you can use the ORD
function with a pointer; the result is an integer which
represents a physical address.

For readers who are familiar with pointers in some other
language, it must be emphasized that in Pascal you cannot set a
pointer to a static variable; pointers can point only to dynamic
variables created by NEW.

Declaring Pointer Variables

A pointer variable is declared by using a pointer type. The
syntax for a pointer type is

pointer type
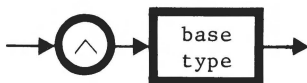
In other words, a pointer type is a base type prefixed with the ^ symbol. The base type specifies the type of variable that a pointer of this type can point to. The base type can be any type except a file type, and there is a special provision: the base type can be an identifier for a variable type that has not yet been declared. This is the only time in Pascal where you can use an identifier that has not yet been declared. Examples:

```
TYPE DATASET = ARRAY [Ø..25] OF REAL;
     RPTR = ^RTYPE;
     RTYPE = RECORD
               LINK: RPTR;
               DATA: DATASET
             END;

VAR INDATA: DATASET;
    FIRST, LAST, CURRENT: RPTR;
...
```

The pointer type RPTR is declared using RTYPE as its base type, even though RTYPE is still undefined. Then RTYPE is declared, and it makes use of the RPTR type in its own declaration, for reasons that will be explained below. The RPTR type is used again in a static variable declaration; FIRST and CURRENT are pointers to point to dynamic variables of type RTYPE.

Using Pointers

The dynamic variable currently pointed to by a pointer is called the object of the pointer. To refer to the object of a pointer, merely refer to the pointer and append the ^ symbol.

For example, suppose that we have the declarations shown above, and the program sets up some values in the array INDATA (perhaps by reading from a file). Now we write

```
NEW(FIRST);
```

which creates a dynamic variable of type RTYPE and sets the pointer FIRST to point to it. The data from the INDATA array can be transferred into the new dynamic variable by writing

```
FIRST^.DATA := INDATA
FIRST^.LINK := NIL
```

Next suppose that the program puts another set of values into
INDATA, and we want to transfer these into a second dynamic
record. If we again use NEW(FIRST), we will lose the pointer to
the first dynamic record; so instead we write

> LAST := FIRST; {The first record is also the last record}

We will leave the value of FIRST unchanged for the rest of the
program; it will always point to the first dynamic record. Now,
for each new dynamic record required by the program, we use the
following:

> {Create new dynamic record, pointed to by last record's
> LINK field:}
> NEW(LAST^.LINK);
> {Make the new record the last record}
> LAST := LAST^.LINK;
> {Transfer data into last record}
> LAST^.DATA := INDATA;
> {Set last record's LINK field to NIL}
> LAST^.LINK := NIL

The first statement creates a new dynamic record and sets the
LINK field of the last record to point to the new record. The
second statement sets the LAST pointer to point to the new
record, and the third transfers the new data into the new
record. The last statement sets the LINK field of the last
record to NIL, so that the program can always tell which dynamic
record is the last in the list.

When all the data has been stored in dynamic records, the result
is a linked list of data arrays, structured as follows:

- The pointer FIRST points to the first record.

- The LINK field of each record except the last points to
  the next record in the list.

- The last record can be identified by the fact that its
  LINK field is NIL.

Suppose that PROCESS is a procedure that takes an ARRAY [$0..25$]
OF REAL as its parameter, and we want to use PROCESS on each of
the data arrays in the dynamic records. We write

```
{Set current pointer to first record:}
CURRENT := FIRST;
{Repeat as long as CURRENT points to something:}
REPEAT
   {Process data of current record:}
   PROCESS(CURRENT^.DATA);
   {Advance to next record:}
   CURRENT := CURRENT^.LINK
UNTIL CURRENT = NIL
```

A general discussion of linked data structures is beyond the
scope of this manual, but the above example shows the essential
idea: each record in a linked data structure contains at least
one pointer, which can point to another record in the structure.
At least one static pointer variable (FIRST in the example)
provides an entry into the structure; after using this pointer to
begin with, other records are accessed by using the pointers
contained in the records.

When a program starts running, pointer variables are not
initialized; they have unknown values until they are set
by NEW or by assignment. A variable reference using a
pointer that has not been set is a reference to some
unknown memory location and can have disastrous results.

The NEW Procedure

As we have already seen, NEW is a built-in procedure that takes a
pointer variable as a parameter. NEW creates a dynamic variable
whose type is the same as the pointer's base type, and sets the
value of the pointer variable to point to the new dynamic
variable. NEW is the only way to create a new pointer value, and
the only way to create a dynamic variable.

So far we have used only the simplest (and most common) way of
calling NEW. The complete syntax is

If there is only one parameter, NEW operates as described in the
previous sections. Constants can be passed if the pointer's base
type is a variant record type; the next section discusses this
technique.

Dynamic Records with Specified Variants

If you are creating numerous dynamic variables of a record type
that has a variant part, you may be able to save space by
specifying the variant for each dynamic variable. This is done
by passing a constant to NEW along with a pointer variable. The
constant must be the same as one of the case labels in the
variant part. The constant specifies which variant is to be used
by this particular dynamic record.

The effect is that the space allocated for the dynamic record is
only large enough for the specified variant. Note that this is
an exception to the usual rule that the space occupied by any
record variable with a variant part is large enough for the
largest variant.

The constant that specifies the variant is not assigned to
the dynamic record's tag field. The program must make
this assignment explicitly after the dynamic record is
created (assuming that there is a tag field).

Also, observe the following cautions when accessing the dynamic
record; if you do not, you may inadvertently destroy information
in another record.

- You should not try to change the variant of the
  record.

- You should not make an assignment to the record as a
  whole. You can freely make assignments to fields
  within the record, as long as the fields do not belong
  to a different variant.

- You should not reference a field that does not belong
  to the variant you have specified.

To see how all this works, consider an example. Suppose that we
have the declarations

```
TYPE  ARRAYSIZE = (SMALL, BIG);
      SMALLARRAY = ARRAY [∅..9] OF REAL;
      BIGARRAY = ARRAY [∅..99] OF REAL;
      VRPTR = ^VREC;
      VREC = RECORD
                LINK: VRPTR;
                CASE WHATSIZE: ARRAYSIZE OF
                  SMALL: (MEANDATA: SMALLARRAY);
                  BIG: (RAWDATA: BIGARRAY)
             END;

VAR INRECORD: RECORD CASE WHATSIZE: ARRAYSIZE OF
                  SMALL: (MEANDATA: SMALLARRAY);
                  BIG: (RAWDATA: BIGARRAY)
                END;
     HEAD, CURRENT: VRPTR;
```

The program proceeds as in the previous example to create dynamic
records and load them with new array values from INRECORD; for
each new dynamic record, this could be done as follows:

```
NEW(CURRENT^.LINK);
CURRENT := CURRENT^.LINK;
CASE INRECORD.WHATSIZE OF
  SMALL: CURRENT^.MEANDATA := INRECORD.MEANDATA;
  BIG: CURRENT^.RAWDATA := INRECORD.RAWDATA
END;
CURRENT^.WHATSIZE := INRECORD.WHATSIZE;
CURRENT^.LINK := NIL
```

This is just like the previous example, except that the CASE
statement is needed to check what size array is contained in
INRECORD and make the appropriate assignment; and of course the
tag field must also be assigned.

But the first statement, NEW(CURRENT^.LINK), causes the new
dynamic variable to be big enough to contain an array of 1∅∅ real
values, even if only an array of 1∅ values is going to be
assigned to it. If the array size in this particular dynamic
variable is not going to change, this is a waste of space.
Instead, we write

```
CASE INRECORD.WHATSIZE OF
  SMALL: BEGIN
            {Create & load small variant of dynamic record:}
            NEW(CURRENT^.LINK, SMALL);
            CURRENT := CURRENT^.LINK;
            CURRENT^.MEANDATA := INRECORD.MEANDATA
         END
  BIG: BEGIN
         {Create & load big variant of dynamic record:}
         NEW(CURRENT^.LINK, BIG);
         CURRENT := CURRENT^.LINK;
         CURRENT^.RAWDATA := INRECORD.RAWDATA
       END
END;
CURRENT^.WHATSIZE := INRECORD.WHATSIZE;
CURRENT^.LINK := NIL
```

Now each dynamic record will occupy only the space it needs to contain the data actually assigned to it.

The syntax for calling NEW allows for more than one constant; this is because a record type can have more than one variant part (nested inside one another). You can specify the variant for each variant part in the record; the sequence of constants in the NEW call is matched with the sequence of variant parts in the record type declaration.

You can supply fewer constants than the number of variant parts; the constants supplied are matched with the variant parts as far as possible, and the remaining variant parts are left unspecified (thus allowing enough space for the largest possible variant).

MEMAVAIL

In working with dynamic variables, it may be necessary to check the amount of available memory before calling NEW to create a new dynamic variable. The MEMAVAIL function returns the number of two-byte words of memory that are guaranteed to be currently available for program data. It can be used as in the following example:

```
TYPE BIGARRAY = ARRAY[0..99, 0..12] OF INTEGER;
...
VAR APTR: ^BIGARRAY;
...
IF 2*MEMAVAIL > SIZEOF(BIGARRAY) THEN NEW(APTR)
ELSE WRITELN('Help, I'm running out of room!');
```

Note that the value returned by MEMAVAIL is multiplied by 2 to convert from words to bytes, since the value returned by SIZEOF(BIGARRAY) is the size in bytes of a variable of type BIGARRAY. The SIZEOF function is described in Chapter 13.

The system, when reading in the directory of an Apple II formatted diskette, allocates approximately 2,000 bytes in program memory for the directory. After the directory is read, MEMAVAIL will indicate that you have 2,000 fewer bytes available. This space is automatically freed for use when you issue a NEW or RELEASE procedure call. You can also cause the space to be freed, without any other effect on your program, by calling MARK.

MARK and RELEASE

Pascal as originally defined by Jensen and Wirth has a procedure called DISPOSE. This procedure is not provided in Apple III Pascal. Instead, the MARK and RELEASE procedures are used for returning dynamic memory allocations to the system. Note that MARK and RELEASE do not provide exactly the same capability as DISPOSE; programs using DISPOSE require some rewriting for use on the Apple III.

When a Pascal program starts running, there is an area of memory that is unused. When a dynamic variable is created by NEW, the space allocated to the dynamic variable is at the beginning of unused memory. Each subsequent dynamic variable is allocated space just after the previous dynamic variable.

Thus while the program is running, there is a sequence of dynamic variables which are physically laid out in the order in which they were created. Note that this is not necessarily the order in which the NEW calls are written in the source program; it is the order in which they are actually executed.

The MARK procedure allows you to mark a particular point in the sequence. After this the program can create more dynamic variables. The RELEASE procedure releases the space occupied by these dynamic variables; the space can then be re-used for dynamic variables created after the RELEASE call.

MARK takes a pointer of any type as its only parameter, and so does RELEASE. The forms are

    MARK (XXX)

    RELEASE (XXX)

where XXX is a pointer variable.

MARK works by making XXX point to the beginning of available unused space. This can be thought of as "marking" the existing space allocation.

RELEASE works by restoring the space allocation that is "marked" by the current value of XXX. That is, RELEASE causes the system to consider that the available unused space begins at the location pointed to by XXX. The effect is to throw away all dynamic variables created since the last time MARK was called with the same variable as its parameter. The chronology is that of the running program, not necessarily the same as the sequence of statements in the program source.

Only the beginning of available space is marked. The end of available space may also change dynamically as the program runs, because of memory usage by the system. The MEMAVAIL function can be used to check the minimum amount of space actually available at any time.

The use of a pointer variable in conjunction with MARK and RELEASE means that a number of marks can be made by calling MARK at different points in the program with different pointer variables. RELEASE can then be called at various other points to release space back to various different marks. The following program fragment is a simple example:

```
VAR MARKER1, MARKER2: ^INTEGER;
...

PROCEDURE ONE;
  BEGIN
  {This procedure creates dynamic variables}
  END;

PROCEDURE TWO;
  BEGIN
  {This procedure creates dynamic variables}
  END;

PROCEDURE THREE;
  BEGIN
  {This procedure creates dynamic variables}
  END;

BEGIN {Main program}
  REPEAT
    {Before calling ONE, mark the existing allocation.}
    MARK(MARKER1);
    ONE;
    {Before calling TWO, mark the existing allocation.}
    MARK(MARKER2);
    TWO;
    {Now we need to release the space used by TWO (but not by
     ONE), before calling THREE.  The following statement
     releases all space dynamically allocated since MARKER2
     was used as parameter for MARK.  The chronology is that
     of the running program.}
    RELEASE(MARKER2);
    THREE;
    {Now we need to release all of the dynamically
     allocated space, before repeating.  The following
     statement releases all space dynamically allocated
     since MARKER1 was used as parameter for MARK.  The
     chronology is that of the running program.}
    RELEASE(MARKER1);
  UNTIL {some condition};
  ...
END.
```

Careless use of MARK and RELEASE can leave dangling pointers, which do not point to any useable data. This can lead to inadvertent destruction of data.

10

Introduction to Files and I/O

This chapter assumes that you are familiar with the Filer
and with the conventions of pathnames. When a pathname is
used in a Pascal program, all conventions are as described
in the Introduction, Filer, and Editor manual (chapter on
the Filer) except that wildcard characters are not
allowed.

This chapter introduces the basic concepts of Pascal files,
including the declaration of files in a program. Next it
presents a brief overview of the I/O facilities of Apple III
Pascal. The remainder of the chapter covers I/O operations on
typed files.

Files

A Pascal file is a special kind of structured variable. A file
variable resembles an array, in that it consists of a sequence of
distinct variable components, all of the same type. However, the
number of components is unknown and they are not accessed by
indexing but via the processes called input and output, or I/O.

External Files

To use a file variable, it must be associated with an external
file. An external file is a peripheral device, or a named
diskette file. The Pascal program does not manage these devices
directly; instead the built-in I/O procedures send requests to
the SOS device drivers, which manage the external devices and
transfer data between external devices and the program in a very
specific manner.

The Pascal programmer need not be concerned with details of
device management; however in Chapter 12 we will see procedures
for direct device-oriented I/O. The communication between the
program and the SOS device drivers is performed automatically by
the built-in input and output procedures.

There are three categories of external files:

   - Character devices, also called non-block-structured
     devices. These include the console, printers, the
     graphics screen, the audio generator, and any other

devices whose input and output are in the form of a
stream of ASCII characters (or individual bytes of
data). For each of these devices there is a SOS
driver, e.g., .CONSOLE, .PRINTER, .SILENTYPE, .GRAFIX,
and .AUDIO. You can use these for I/O by giving the
driver name as the pathname (you can also use Pascal
unit names or numbers; see Appendix J).

- Block-structured devices. These are devices, such as
  diskette drives, that store blocks of data. Other
  mass-storage devices would also be in this category.
  These also have their own SOS drivers (.D1, .D2,
  etc.). Chapter 12 describes methods of using these
  devices directly; more commonly they are considered as
  containers for stored files.

- Stored files on block-structured devices, e.g.,
  diskette files. These are the files that appear in the
  directories for diskettes. You can use these for I/O
  by giving a pathname that specifies a diskette volume
  name and a local filename. (You can also use the SOS
  device driver name, a Pascal unit name, or a Pascal
  unit number instead of the diskette volume name; see
  Appendix J.)

In this chapter and the next we are concerned with character
devices and named files. For the most part, these two kinds of
files are handled identically; the few cases where there are
differences in special-character handling are described as they
come up.

File Variables

The most important feature of a file variable is that its values
are not generally in memory. Instead, they exist outside the
program as the contents of an external file. However, an
1,1Ø0-byte area of memory is allocated for every declared file in
a currently active procedure.

A file variable is declared along with other variables, using a
file type. In this chapter and the next we are concerned with
typed files; "block" files are covered in Chapter 12. The syntax
of a file type (for a typed file) is

file type (for typed file)
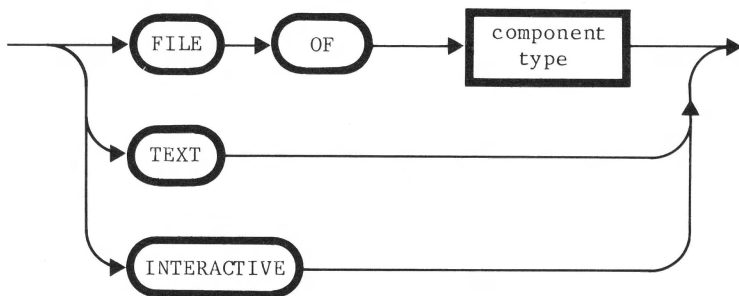
where the component type can be any type except a file type.
TEXT and INTERACTIVE are built-in file types: TEXT is exactly
equivalent to FILE OF CHAR, and INTERACTIVE is a special type
described in the next chapter.

External diskette files also have "types," which are
independent of Pascal file variable types. One of these
external file types is called "Textfile". Note that the
file variable type TEXT and the external file type
"Textfile" are two different things; a file variable of
type TEXT may or may not correspond to an external file of
type "Textfile." See Chapter 12 for a description of the
external "Textfile" type and what it means to a Pascal
program.

Remember that the file variable type determines how the
Pascal program will interpret the file's contents. The
external type of the file determines how the data are
actually stored and formatted on the diskette. As a
general rule, the external file type makes no difference
to a Pascal program that uses the I/O mechanisms described
in this chapter and in Chapter 11. The external file type
"Textfile" affects random access with the SEEK procedure
(explained further on in this chapter) and in block I/O
and device-oriented I/O (see Chapter 12).

The components of a file variable are usually called logical
records or simply "records," although their type is not
necessarily a record type. For example, the declarations

```
VAR INTVALS: FILE OF INTEGER;
    REALVALS: FILE OF REAL;
    COMPVALS: FILE OF RECORD
                    I: INTEGER;
                    R: REAL
              END;
```

create three file variables. INTVALS is a file variable whose
records are integer values, and REALVALS is a file variable whose
records are real values. COMPVALS is a file variable whose
records are actually of a record type, with an integer value and
a real value in each record.

Using a File

This chapter goes into I/O mechanisms in detail, but first we
present a brief description of how a program can store values in
a diskette file and how they can be retrieved later.

The RESET and REWRITE procedures, described in detail further on,
serve to associate a file variable with a specific pathname; if
the pathname refers to a diskette drive or a named diskette, the
file variable can be associated with a specific named file on the
diskette. Once this has been done, the file is said to be "open"
and its records can be accessed. For example, suppose that we
have the declarations shown above and that a diskette drive
contains a diskette named /VALUES. If we write

        REWRITE (INTVALS, '/VALUES/INTEGERS.DATA')

the effect is to associate the file variable INTVALS with the
diskette file /VALUES/INTEGERS.DATA. Because we opened the file
with REWRITE instead of RESET, SOS automatically creates a
diskette file named INTEGERS.DATA on diskette /VALUES. (If there
is already a diskette file by that name, SOS will later delete
either the old diskette file or the new one, depending on how the
program "closes" the file.)

REWRITE also prepares for I/O by defining a buffer variable
called INTVALS^. This is an integer variable, since INTVALS is a
FILE OF INTEGER. Suppose that the program contains an integer
variable XYZ, and we want to write the value of XYZ into INTVALS
as the first record of the file variable. We write

        INTVALS^ := XYZ;
        PUT(INTVALS)
```

The first statement assigns the value of XYZ to the buffer
variable, and the PUT procedure (via SOS) writes the value out as
the first record of the diskette file /VALUES/INTEGERS.DATA.    The
next PUT(INTVALS) call will write out the buffer variable to the
second record of /VALUES/INTEGERS.DATA, and each subsequent
PUT(INTVALS) writes the next record in sequence.

When the program is through using the INTVALS file, it closes
it:

    CLOSE(INTVALS, LOCK)

The LOCK option tells SOS to make the new /VALUES/INTEGERS.DATA
diskette file permanent.

Now we have a permanent diskette file containing some unspecified
number of integer values.  Suppose that another program needs to
access this data.  Again we must declare a file variable of
appropriate type:

    VAR INDATA: FILE OF INTEGER;

Although this file variable has a different name from the one
used previously, it can of course be associated with the same
diskette file.  In the new program, we open the file with RESET
instead of REWRITE:

    RESET(INDATA, '/VALUES/INTEGERS.DATA')

Unlike REWRITE, RESET requires that the external file already
exist.  (If it does not, the program is halted with an error
message.)  RESET creates a buffer variable of type integer (named
INDATA^) and, unlike REWRITE, it loads the buffer variable with
the first value in the diskette file.  Suppose that we want to
assign this value to an integer variable named ABC, and then have
the next value from the diskette file loaded into the buffer
variable.  We write

    ABC := INDATA^;
    GET(INDATA)

The effect of GET is always to load the next record from the
external file into the buffer variable (by issuing a request to
SOS).

Note the parallelism between output and input. For output we opened the file with REWRITE, and then each value was written out by

- Using the buffer variable (by assigning the value to it).

- Calling PUT with the name of the file variable as its parameter.

For input we opened the file with RESET, and then each value was read in by

- Using the value of the buffer variable (assigning it to another variable in this case).

- Calling GET with the name of the file variable as its parameter.

Before proceeding to a detailed description of the built-in I/O procedures for typed files, we give an overview of all the file I/O facilities provided in Apple III Pascal.

## Overview of Apple III Pascal I/O Facilities

Apple III Pascal I/O facilities fall into four different categories:

- Device-oriented I/O (covered in Chapter 12): The procedures UNITREAD, UNITWRITE, UNITBUSY, UNITWAIT, UNITSTATUS, and UNITCLEAR 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 SOS pathnames, directories, etc., but merely use unit numbers and (for diskette drives) block numbers.

- Block I/O (covered in Chapter 12): The BLOCKREAD and BLOCKWRITE functions provide I/O for untyped files. They make use of SOS pathnames and directories, but they consider a file to be merely a sequence of 512-byte blocks--not a sequence of logical records of a particular type.

- Text I/O (covered in Chapter 11):  The READ, READLN,
  WRITE, and WRITELN procedures provide text-oriented I/O
  for files of characters.  The PAGE procedure writes a
  form feed control character into a file of characters.
  The EOLN function provides an indication of when the
  end of a text line has been reached.

- Typed file I/O (covered in the remainder of this
  chapter):  The GET, PUT, and SEEK procedures treat a
  file as a sequence of logical records.  GET and PUT
  provide transfers between individual file records and
  the file's buffer variable, and SEEK causes the next
  GET or PUT to access a specified record.  The EOF
  function provides an indication of when the end of the
  file has been reached.

# *Typed File I/O*

This section is concerned with typed files--that is, files that
are considered as a sequence of logical records.  Typed files are
declared as shown at the beginning of this chapter.  Such file
variables are generally associated with diskette files, and all
of the examples given here use diskette files.  The essential
mechanisms for I/O operations with typed files are the GET and
PUT procedures.

For files where the logical records are characters (types FILE OF
CHAR, TEXT, and INTERACTIVE), Pascal also provides specialized
text-oriented I/O as explained in the next chapter.  Character
files can be accessed either with the text I/O procedures or with
GET and PUT as described below; however there are some special
considerations for INTERACTIVE files which are discussed in the
next chapter.

If an I/O operation is unsuccessful, the system will
normally terminate program execution with an error
message.  However, there is a Compiler option to disable
this feature, as described in Appendix F.  The IORESULT
function can then allow the program itself to check on the
status of the most recent I/O operation and take
appropriate action.

# The REWRITE Procedure

This procedure opens a new file--i.e., it creates an external
file, associates it with a file variable, and creates a buffer
variable. The file can then be used for input and output.
REWRITE is normally used for output. The form for calling
REWRITE is

        REWRITE ( FILEID, PATHNAME )

where FILEID is the identifier of a file variable, and PATHNAME
is an expression with a string value. The string value is the
pathname of the external file.

If the pathname specifies a character device, then the file is
simply opened. If the pathname specifies a diskette file,
REWRITE creates a new file and opens it. If the pathname is the
same as that of an existing diskette file, a new diskette file is
created with the same name; one or the other will later be
deleted as described below under CLOSE.

If the file is a diskette file and is already open--i.e., if the
file identifier corresponds to a currently open file--an I/O
error occurs (see IORESULT below). The file remains open.

However, any external file that has been opened can legally be
opened again via REWRITE with a different file variable.

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

# The RESET Procedure

This procedure, like REWRITE, opens a file which can then be used
for input and output. RESET is normally used for input. Unlike
REWRITE, RESET requires that the external file already exist.
The syntax for calling RESET is:



Note that the string expression may be omitted; this is explained
below. The string value is the pathname of the external file.

If there is no file with this name, an I/O error occurs (see IORESULT, below).

If the file is a diskette file and is already open--i.e., if either the file identifier or the diskette-file pathname corresponds to a currently open file--an I/O error occurs (see IORESULT below). The file remains open. However, if a non-diskette file has been opened, it can legally be opened again with a different file variable.

RESET automatically performs a GET action--that is, it loads the first record of the file into the file's buffer variable, and the first explicit GET or PUT in the program will access the second record. Thus, resetting a non-interactive file to the console requires the user to type a character when the RESET is executed. If the file is INTERACTIVE (see next chapter) no GET is performed.

Note that RESETing 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. In this case, use REWRITE to open the file.

When a file opened with RESET is used for output, only the file records actually written are affected.

RESET can also be called without a pathname if the file is already open. The effect is simply to "reopen" the file, using the same external file. This allows the program to go back to the beginning of the file and re-access records. The buffer variable contains the value of the first record, and the next explicit GET or PUT will move the buffer variable to the second record. In the case of the console, the primary use of RESET without a pathname is to clear end-of-file.

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

## The CLOSE Procedure

This procedure closes a file which was previously opened with RESET or REWRITE. The syntax for calling CLOSE is

where the option identifier, if used, may be any one of the
following predefined identifiers.  If no option identifier is
used, the effect is the same as using the NORMAL option.

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

LOCK -- If the external file is a diskette file and was
opened with REWRITE, it is made permanent in the
diskette's directory; otherwise a NORMAL close is
done.  If the file was opened with a REWRITE and the
pathname matches an existing diskette file, the old
file is deleted.

WPROTECT -- Same as LOCK, but the external file is made
write-protected.  Any future attempt to write to the
file, delete it, or rename it (either with a file I/O
procedure or from the command level) will cause an I/O
error (see IORESULT below).  However, note that the
file is not protected against the UNITWRITE procedure
(see Chapter 12).

UNPROTECT -- Same as LOCK, but write-protection is
cancelled if the file was opened with a RESET.

PURGE -- If the external file is a diskette file, it is
deleted from the directory (unless write-protected).
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 external
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 record accessed is thrown
away.

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

If a program terminates with a file open (i.e., if CLOSE is omitted), the system automatically closes the file with the NORMAL option.

If you open an existing file with RESET and modify the file with any write operation, the contents are immediately changed no matter what CLOSE option you specify.

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

## Effect of CLOSE Options

The following diagrams show the result of closing a file which was opened with REWRITE or RESET, depending on the CLOSE option selected.

## diagram:   opened with rewrite

**CLOSE Option / File**

Opened with REWRITE

| | NORMAL | LOCK | WPROTECT | UNPROTECT | PURGE | CRUNCH | |
|---|---|---|---|---|---|---|---|
| No Old File | Pathname not entered; contents thrown away | Pathname entered; contents kept | Pathname entered & protected; contents kept | Pathname entered; contents kept | Pathname not entered; contents thrown away | Pathname entered; contents truncated and kept | |
| Old Unprotected File Exists | Contents thrown away | Contents kept | Pathname protected; contents kept | Contents kept | Contents thrown away | Contents truncated and kept | New file results |
| | Contents kept | Contents deleted | Contents deleted | Contents deleted | Contents kept | Contents deleted | Old file results |
| Old Protected File Exists | Contents thrown away | Run-time error | Run-time error | Run-time error | Contents thrown away | Run-time error | New file results |
| | Contents kept | | | | | | Old file result |

Example:   If an old unprotected file is opened with REWRITE and closed with CRUNCH, the new file contents are truncated and kept while the old file contents are deleted.

## diagram:   opened with reset

**CLOSE Option / File**

| | NORMAL | LOCK | WPROTECT | UNPROTECT | PURGE | CRUNCH |
|---|---|---|---|---|---|---|
| No Old File | Run-time I/O error upon RESET; file never opened | | | | | |
| Old Unprotected File Exists | Contents kept | Contents kept | Pathname protected; contents kept | Contents kept | Pathname & contents deleted | Contents truncated & kept |
| Old Protected File Exists | Contents kept | Contents kept | Contents kept | Pathname unprotected; contents kept | Run-time error | Run-time error unless at EOF |

To determine if an existing file is write-protected:

- reset the file;
- do a read;
- reset the file again (to go to the beginning);
- try to write the information you just read.

If the file was write-protected, the write will fail; otherwise the write will succeed and leave the file unchanged.

To remove the protection from a write-protected file, reset the file and then close it with the UNPROTECT option.

## 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 syntax for calling EOF is



If no file identifier is used, the predefined file INPUT (see next chapter) is assumed.

EOF is false immediately after the file is opened if the file has any records in it; EOF is true on a closed file. Whenever EOF is true for a particular file, the value of the file's buffer variable is undefined.

After an input operation, EOF is true if the operation attempted to access a record that is after the end of the file. When this is the case, the value of the file's buffer variable is not defined.

After an output operation, EOF is true if the file cannot be expanded to accommodate the output (because of limited diskette space, for example). However, note that this will generally be considered an I/O error, causing the program to halt with an

error message before EOF can be checked.

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

## *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 file, which must be open.

GET moves the contents of the next external file record into the
file's buffer variable.  Note that immediately after RESET, the
contents of the first external file record are already in the
buffer variable and the "next" record is the second record.

There is one special case:  when GET is used with a file
of type TEXT or INTERACTIVE and a "return" character
(ASCII 13) is read, the character is converted to a space
(ASCII 32).  Also, the function EOLN (see next chapter)
will then return TRUE.

See the last section of this chapter for a note on special
handling of control characters.

PUT puts the contents of the file's buffer variable into the next
external file record.  If the file was opened with REWRITE, this
means that the "next" record is created; if it was opened with
RESET, the "next" record already exists and is overwritten.

Note that when a non-INTERACTIVE file is opened with RESET, there
is apparently no way for PUT to access the first record of the
external file.  This is because immediately after the RESET the
"next" record is the second one.  The SEEK procedure (see below)
provides a way around this limitation.

After a GET or PUT, the next GET or PUT with the same file
identifier will access the next external file record in
sequence.

With diskette files, the actual physical diskette access may not
occur until the next time the physically associated block of the
diskette 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 component type REAL, with the pathname /WTS/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 in the next
chapter.

```
PROGRAM MAKEFILE;

VAR F: FILE OF REAL;
    I: INTEGER;

BEGIN
  {Open with REWRITE since this is a new file.}
  REWRITE(F, '/WTS/REAL.DATA');
  {Read 1Ø numbers and store them in the file.}
  FOR I:=1 TO 1Ø DO BEGIN
    {Put a prompt on the screen.}
    WRITE('-->');
    {Read number from keyboard into buffer variable.}
    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, '/WTS/REAL.DATA');
  {First number is in F^.  While EOF remains false, display
   each number and get the next.}
  WHILE NOT EOF(F) DO BEGIN
    {Display the current number on the screen}
    WRITELN(F^);
    {Get the next number}
    GET(F)
    {If there was no next number, EOF is now true.}
  END;
  {Close the file}
  CLOSE(F)
END.
```

Note that these programs offer no flexibility as to the pathname
of the file. The next example will show how to let the program's
user specify the pathname of the external file to be used.

The IORESULT Function

This built-in function takes no parameters and returns an integer
value which reflects the status of the last completed I/O
operation:

Ø	No error; normal I/O completion
2	Bad unit number
3	Illegal operation (e.g., read from .PRINTER)
5	Lost unit--no longer on line
6	Lost file--file is no longer in directory
7	Illegal pathname
8	No room--insufficient space on diskette
9	No unit--unit is not on line
1Ø	No such file in specified directory
11	Duplicate pathname
12	Attempt to open an already open file
13	Attempt to access a closed file
14	Bad input format--error in reading number
15	Ring buffer overflow--input arriving too fast
16	Write-protect error--diskette is protected
19	Too many files open for system to handle
48..63	(SOS) Device-specific error
64	Device error--bad address or data on diskette

65	(SOS) Too many character files open
66	(SOS) Too many block files open
73	(SOS) Directory full
74	(SOS) Incompatible file format
81	(SOS) Volume format neither SOS nor Apple II
84	(SOS) Out of memory for SOS system buffer
87	(SOS) Duplicate volume error

The (SOS) notation indicates an error reported by SOS as opposed to the Pascal system. There are some other SOS I/O errors besides those shown above; see Appendix J for a complete table.

In normal operation, the Compiler will generate code to perform run-time checks after each I/O operation except UNITREAD or UNITWRITE. This causes the program to be halted with a run-time error message on a bad I/O operation. Therefore if you want to check IORESULT with your own code in the program, you must disable run-time I/O checking by using a <u>Compiler option</u>.

Compiler options are commands to the Compiler, embedded in the program text. The complete set of options is described in Appendix F, but here we are concerned with just two: the text

    {$IOCHECK-}

disables run-time I/O checking, and the text

    {$IOCHECK+}

enables it. The following sample program illustrates the use of these options. It allows the user to add new records to a file containing information about people's birthdays. If the external file specified by the user does not already exist, the program detects this fact by using IORESULT. If IORESULT does not return a $\emptyset$ value, the program uses REWRITE instead of RESET to create the external file.

```
PROGRAM ADDRECORDS;
  {Adds records to a birthday file.  Creates the file if
   necessary.}
VAR
  INCHAR: CHAR;
  PATHNAME: STRING;
  BIRTHDAYS: FILE OF RECORD
                    NAME: STRING[2Ø];
                    DAY, MONTH: INTEGER;
                  END;
BEGIN

  {Obtain pathname.}
  WRITE('Enter pathname: ');
  READLN(PATHNAME);

  {Use RESET to preserve existing contents of file; if
   it doesn't exist, then use REWRITE to create it.}
  {$IOCHECK-} {This turns off I/O error checking, so
          if RESET doesn't work the program doesn't
          get halted.  Then it can use IORESULT to
          check for error.}
  RESET(BIRTHDAYS, PATHNAME);
  {$IOCHECK+} {This turns I/O error checking back on.}
  IF IORESULT<>Ø THEN REWRITE(BIRTHDAYS, PATHNAME);

  {Go to end of file with repeated GETs.}
  WHILE NOT EOF(BIRTHDAYS) DO GET(BIRTHDAYS);
```

```
{Repeat the following until user doesn't want to make
any more new records.}
REPEAT
  {Find out if user wants to make a new record.}
  WRITE('Make a new record? (Type y or n): ');
  READ(INCHAR);
  IF INCHAR IN ['y', 'Y'] THEN BEGIN
    {Create the new record}
    WITH BIRTHDAYS^ DO BEGIN
      WRITELN;
      WRITE('Enter name: ');
      READLN(NAME);
      WRITE('Enter day: ');
      READLN(DAY);
      WRITE('Enter month: ');
      READLN(MONTH)
    END;
    {PUT the new record into file.}
    PUT(BIRTHDAYS)
  END {of "if inchar in ['y', 'Y']"}
UNTIL NOT (INCHAR IN ['y', 'Y']);

CLOSE(BIRTHDAYS, LOCK) {Must use LOCK in case REWRITE
                        was used to open.}
END.
```

Be careful if you disable I/O error checking with the
{$IOCHECK-} option. You will not be informed of the
failure of any I/O operation that is performed while I/O
error checking is disabled.

Every I/O operation potentially changes IORESULT.
Therefore, if you want to write out the value of IORESULT
for a previous I/O operation, you should save it in an
auxiliary variable and then write the auxiliary variable.

Random Access

Introduction

So far, we have only discussed <u>sequential access</u> to files: after
accessing any particular file record, the next record accessed is
the next one in sequence. In <u>random access</u>, a program can access
any existing file record at any time, without regard to the
sequence of records in the file. Note that this means a given
record can be accessed repeatedly, without the need to RESET the
file. For example, a program might first read all records of a
diskette file sequentially, and then alter the 8th, 2nd, 1Øth,
and 5th records in that order.

This is done by using the SEEK procedure with record numbers.
The sequence of records in a file can be regarded as a numbered
sequence: the first record in the file is record number Ø, the
next is record number 1, and so forth.

Incidentally, the terms "random file" and "sequential
file" are commonly used but misleading. Random and
sequential are two methods for accessing files--not two
kinds of files. Any typed file can be accessed randomly
or sequentially or both ways. Whether it makes sense to
use a particular kind of access depends on the program and
on the contents of the file records.

The SEEK Procedure

The SEEK procedure allows the program to access any specified
record in a file, and this provides random access to file
records. The form for calling SEEK is

    SEEK (FILEID, RECNUM)

where FILEID is the identifier of a file and RECNUM is an
expression with an integer value that specifies a record number
in the file. Note that records in files are numbered from Ø.

SEEK affects the action of the next GET or PUT from/to the file,
forcing it to access the specified file record instead of the
"next" record. SEEK does not affect the file's buffer variable.

The file should be a file on a diskette or other block-structured device. It should not be a character device, nor should it be declared as a file of type TEXT, INTERACTIVE, or FILE OF CHAR.

If the file is a character device or is of the type TEXT, INTERACTIVE, or FILE OF CHAR, SEEK does nothing.

A GET or PUT must be executed between SEEK calls since two SEEKs in a row may cause unpredictable results. Immediately after a SEEK, EOF will return false; a following GET or PUT will cause EOF to return the appropriate value.

The record number specified in a SEEK call is not checked for validity. If the number is not the number of a record in the file and the program tries to GET the specified record, the value of the buffer variable becomes undefined and EOF becomes true. In fact, this is a way to check the validity of a record number: SEEK it and then do a GET. If EOF is false after the GET then the number was valid.

But if SEEK is called with an invalid record number and the program tries to do a PUT to the specified record, the result of the PUT is unpredictable and nay cause an I/O error. Therefore if there is any doubt about the validity of a record number, the program should make sure that it is valid.

The standard system library file SYSTEM.LIBRARY contains a unit named PASCALIO. The SEEK procedure cannot be executed unless this unit is available in a library at the start of execution.

The following sample program demonstrates the use of SEEK to randomly access and update records in a file. It uses a file of birthday information created by the program shown in the previous section, and it allows the user to correct existing records in this file.

```
PROGRAM RANDOMACCESS;
{Allows update of any selected record in birthday file.}
VAR
   RECNUMBER: INTEGER;
   FNAME: STRING;
   BIRTHDAYS: FILE OF RECORD
                     NAME: STRING[2Ø];
                     DAY, MONTH: INTEGER;
                 END;

BEGIN
   {Obtain filename.}
   WRITE('Enter filename: ');
   READLN(FNAME);
   {Use RESET to preserve existing contents of file; if
    it doesn't exist, complain and stop the program.}
   {$IOCHECK-} {This turns off I/O error checking.}
   RESET(BIRTHDAYS, FNAME);
   {$IOCHECK+} {This turns I/O error checking back on.}
   IF IORESULT<>Ø THEN BEGIN
     WRITELN('File not found.');
     EXIT(PROGRAM)
   END;

    {Repeat the following until user types a negative record
     number, or until there is not enough diskette space for
     new records.}
   REPEAT
     {Obtain record number; fall through to end if user types
      negative record number.}
     WRITE('Enter record number (negative to quit): ');
     READLN(RECNUMBER);

     IF RECNUMBER >= Ø THEN BEGIN

        {GET the specified record}
        SEEK(BIRTHDAYS,RECNUMBER);
        GET(BIRTHDAYS);
```

```
      {If record number was invalid, EOF(BIRTHDAYS) will now
      be true, so skip the rest and get another number.}
      IF NOT EOF(BIRTHDAYS) THEN BEGIN
        {Update the record}
        WITH BIRTHDAYS^ DO BEGIN
          WRITELN('Name is ', NAME);
          WRITE('Enter correct name: ');
          READLN(NAME);
          WRITELN('Day is ', DAY);
          WRITE('Enter correct day: ');
          READLN(DAY);
          WRITELN('Month is ', MONTH);
          WRITE('Enter correct month: ');
          READLN(MONTH)
        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 BIRTHDAYS^ }

        SEEK(BIRTHDAYS,RECNUMBER);

        {PUT updated record into file.}
        PUT(BIRTHDAYS)
      END {of "if not eof(birthdays)"}
    END {of "if recnumber >= Ø"}
  UNTIL RECNUMBER < Ø;
  CLOSE(BIRTHDAYS)
END.
```

Special Handling of Control Characters with GET and PUT

Files of type TEXT or INTERACTIVE are usually accessed with the text I/O procedures described in the next chapter. When GET and PUT are used with files of type TEXT and INTERACTIVE, there are some special considerations for three special control characters, namely

    The CR (or "return" or CTRL-M) character, ASCII 13
    The DLE (or CTRL-P) character, ASCII 16
    The CTRL-C (or ETX) character, ASCII 3

Because of the special handling of these characters, described below, GET and PUT should not be used for control-character communication with device drivers. Instead, use UNITREAD and UNITWRITE as described in Chapter 12.

The CR character is used in files of characters to mark the end of a line. The special handling is as follows:

- As already mentioned, when this character is read by GET it is converted to a space (ASCII 32).

- When PUT writes the CR character to a diskette file, there is no special action; the CR character is simply written into the file.

- When PUT writes the CR character to a character device a linefeed character (ASCII 1∅) is automatically written immediately after the CR.

The DLE character is used in textfiles as the first character of a two-character code to represent indentation at the beginning of a line or a sequence of spaces anywhere on a line; this is called a DLE-blank code and is explained in detail at the end of Chapter 12. The special handling is as follows:

- When the DLE character is read by GET, it and the following character are converted to a sequence of spaces (assuming that a sequence of GETs is used for reading). The exception is when GET is used to read from the console; in this case the DLE and the following character are simply read like any other characters.

- When PUT writes the DLE character to a diskette file, there is no special action; the DLE character and the following character are simply written out. However when output is to a character device, the DLE character and the next character following it are converted into a sequence of spaces (see Chapter 12 for details).

The CTRL-C character is used with character devices as an "end-of-text" indicator. The special handling is as follows:

- When the CTRL-C character is read from a diskette file
  by GET, no special handling occurs; the CTRL-C is read
  like any other character.

- When the CTRL-C character is read from a character
  device by GET, it is converted to a space.   EOF and
  EOLN (see next chapter) will return TRUE.

- When PUT writes the CTRL-C character, there is no
  special action; the CTRL-C character is simply written
  out like any other character.   However when output is
  to the console, this has a special effect as a screen
  control code.   In general, do not write any control
  character to the console unless you pay careful
  attention to the section on "Screen Control Codes" in
  the Standard Device Drivers Handbook.

11

Text I/O

Introduction

In addition to PUT, GET, and EOF, Apple III Pascal provides the standard procedures READ, READLN, WRITE, WRITELN, EOLN, and PAGE, collectively known as the text I/O procedures.

WRITE, WRITELN, READ, and READLN have already been introduced informally in examples. WRITE and WRITELN have the effect of writing out text. In the examples, they have been used to write text on the screen. The only difference between the two is that WRITE writes only the specified text, while WRITELN writes the specified text so that the next character written will start a new line.

READ and READLN have been used in examples to read text from the keyboard. The difference between the two is that READLN advances to the beginning of the next line after reading, and READ does not.

The EOLN function is used to detect an end-of-line in text input, and the PAGE procedure places a form-feed (ASCII 12) in text output.

These procedures have capabilities beyond what has already been shown; complete details are given in this chapter. Generally, they give Pascal programs a convenient way to treat a file of characters as a sequence of characters organized into lines; the characters within a line can be treated as individual character values or as strings, or interpreted as numeric values (INTEGER or REAL). To support these capabilities, Apple III Pascal has some special features for files of characters.

Character Files

The text I/O procedures can only be used with files that are declared as character files. A character file is any file whose components are declared to be of type CHAR. Thus the type of a character file can be any one of the following:

```
FILE OF CHAR
TEXT
INTERACTIVE
```

Note that this definition is not the same as the SOS usage of the term "character file." In this manual, the term is always meant to apply to the Pascal types listed above.

As previously noted, the built-in type TEXT is exactly equivalent to FILE OF CHAR. In the remainder of this chapter, we will refer only to the types TEXT and INTERACTIVE; remember that FILE OF CHAR is the same thing as TEXT.

Various parts of the system that deal with files of characters (such as the Editor and the Compiler) are designed to use files of the external type "Textfile". For most purposes, it is therefore recommended that you use the "Textfile" type for any character files created by your programs. This means that when you create a character file with REWRITE, the pathname should end with the .TEXT suffix.

For the purposes of this chapter, however, the distinction between the "Textfile" type and other external types is unimportant; the text I/O procedures make the special formatting of a "Textfile" invisible to the Pascal program. In particular, this means that Pascal programs can process "Asciifiles" created in BASIC.

INTERACTIVE Files

The difference between TEXT and INTERACTIVE files is in the way they are handled by the RESET, READ, and READLN procedures.

When a Pascal program READs characters from an existing 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 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 <u>not</u> 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
begins 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 a character device, it is not
possible to perform a GET on it until the device has a character
ready for input. If RESET tried to do a GET from the keyboard,
for example, the program would then have to wait until a
character was typed. With the INTERACTIVE type, the program
doesn't perform a GET until it is executing a READ or READLN.

Therefore, the type INTERACTIVE is normally used for text I/O
with any character device.

Built-In Files

For convenience in performing console I/O, three files are
predeclared. Their identifiers are OUTPUT for output to the
screen, and INPUT and KEYBOARD for input from the keyboard. All
three of these files are of type INTERACTIVE, and all three are
automatically opened via RESET when the Pascal program begins
executing.

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 displayed. This allows a Pascal program to
have complete control over the screen's response to characters
typed by the user.

When the EOF function is used with these files, it behaves in a
special way: EOF(KEYBOARD) and EOF(OUTPUT) are <u>never</u> true, and
EOF(INPUT) only becomes true when CTRL-C (ASCII 3) is typed and
remains true until a RESET is executed.

Using the Procedures and Functions

The operation of the text I/O procedures is very straightforward
when they are used consistently to treat a file as a sequence of

text entities for either input or output. However, difficulties
may arise in the following unusual cases:

- When you explicitly use GET, PUT or the file buffer
  variable in combination with text I/O procedures on the
  same file variable. (As it turns out, there is rarely
  any reason to do this.)

- When you mix reading and writing operations on the same
  file variable. (Instead, you can use one file variable
  for input and another for output.)

The reason is that the text I/O procedures themselves use GET and
PUT in complicated ways, and therefore the exact position in the
file can become a question. Specifically, it is sometimes hard
to be sure about the following points:

- Exactly what character is the "next" record in a file
  after a READ or READLN.

- Exactly when EOLN or EOF becomes true.

- Whether the file buffer variable contains the "current"
  character or the "next" character.

The exact rules are given at the end of this chapter. Note that
they depend, in some cases, on whether the file variable is of
type TEXT or INTERACTIVE and also on what type of variable is
used with READ or READLN.

The EOLN Function

EOLN is defined only for character files. This function returns
a BOOLEAN value to indicate whether the end of a text line has
been reached in the input from a specified character file. The
syntax is

If no file identifier is given, INPUT is assumed.

EOLN returns true on a closed file. It returns false immediately after the file is opened, except in the special case of a file of type TEXT that begins with the RETURN character.

As a rule, EOLN returns true whenever EOF is true. There is only one exception: EOLN never returns true after a READLN operation on an INTERACTIVE file (regardless of whether EOF is true).

For more details on the behavior of EOLN after a READ or READLN, see the descriptions below.

The READ Procedure

This procedure can be used only on character files. It allows characters, strings, 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 syntax for calling READ is

where the file identifier must be that of an open character file. If the file identifier is omitted, the predeclared file INPUT is assumed. Each variable reference may refer to a variable 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. For example,

    READ(TSTFIL,INCHAR)

(where TSTFIL is a character file and INCHAR is a CHAR variable) reads the next character from TSTFIL and assigns its value to INCHAR.

Special characters are handled as follows:

- Whenever the RETURN character (ASCII 13) is read, it is converted to a space (ASCII 32).

- A DLE-blank code found in a "Textfile" is converted by READ to a sequence of spaces. The exception is when READ is used for input from the console; in this special case a DLE is not converted but simply read like any other character.

- When the CTRL-C character (ASCII 3) is read from a diskette file, there is no special handling; the CTRL-C is read like any other character.

- When the CTRL-C character is read from a character device, it is converted to a space and causes EOF and EOLN to return true.

- 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 also whenever EOF is true. (A "text character" here

means a character that is not the RETURN character.)

– For an INTERACTIVE file, EOF is not true until the
program attempts to read past the last character in the
file, or until the CTRL–C character is read from a
character device. EOLN is not true until the RETURN
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
detect the end of an input line, 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 STRING Variable

For a variable of type STRING, READ reads all the characters up
to the end of the line or until a CTRL–C is read from a character
device. The RETURN character, or CTRL–C from a character device,
is not read. If this is directly followed by another READ with a
string variable, the result is to read nothing, since there are
no more characters on the line. This can lead to an infinite
loop if not handled correctly. We therefore recommend that you
use READLN with a string variable; this has the same effect as
READ except that READLN skips to the beginning of the next line
after reading.

READ with a Numeric Variable

For a variable of type INTEGER, LONG INTEGER, or REAL, READ
expects to read a string of characters which can be interpreted
as a numeric value of the same type. Any space or RETURN
characters preceding the numeric string are skipped. The string
includes all characters up to the first character that does not
fit the syntax for an integer or real value representation
(depending on the variable), or up to the end of the file.

The string is converted to a numeric value and the value is
assigned to the variable. For example,

    READ(X)

(where X is a REAL variable) reads characters from the INPUT file
(the keyboard). It first skips any spaces or RETURNs; then it

reads as many subsequent characters as it can interpret as part of a real number. Spaces, RETURNs, and end-of-files, will frequently delimit reals in a text file, since use of WRITE and WRITELN will produce just such files. The interpreted value is assigned to X.

If READ(X) does not find a numeric string after skipping spaces and RETURNs, the program is halted with an error message. (For further information, refer to Appendix E.)

If nothing but spaces and/or RETURNs are found before the end-of-file, the result depends on whether READ is looking for a REAL or an INTEGER value. If READ is looking for an INTEGER value, a value of Ø is read; but if READ is looking for a REAL value, the program is halted with an error message.

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.

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

The standard system library file, SYSTEM.LIBRARY, contains a library unit called PASCALIO. READ or READLN with a REAL variable cannot be executed unless this unit is available in a library at the start of execution.

When READ is looking for a REAL value:

- any number of characters can be read for a REAL variable, but at most 9 significant digits are converted.

- READ will use decimal-to-binary conversion methods that are slightly different from those used for REAL values written in the program as constants.

- READ will accept '.NaN.' (any combination of
  upper- and lower-case characters) as a REAL
  value which is "not a number".

- READ will accept a string of two or more plus
  signs as positive infinity and two or more minus
  signs as negative infinity.

See Appendix E for further information.

With the exception of NaNs and infinities, the syntax of a
real number entered by READ is the same as the syntax of a
real number appearing in a program.

Apart from these differences, the rules are the same as
those for constants of type REAL. In particular, note
that if there is a decimal point it must be preceded and
followed by numeric digits.

The READLN Procedure

This procedure may be used only on character files. It allows
line-oriented reading of characters, strings, and numeric
values. The syntax for calling READLN is

where the file identifier must refer to an open character file.
If the file identifier is omitted, INPUT is assumed. Each
variable 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 RETURN). 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. If EOF is already true, an I/O error will occur.

READLN with a STRING variable reads all the characters up to but
not including the RETURN character, then skips to the next line.
Thus repeated READLNs with a STRING variable have the effect of
reading successive lines of the file as strings.

Unlike READ, READLN can be called with no variable references at
all. READLN with no variable references simply skips to the
beginning of the next line in the input file.

One of the most common uses of READLN with a STRING variable is
to read a string of characters from the keyboard. In the
following example, READLN is used to read a pathname typed by the
user:

```
    PROGRAM MAKEFILE;
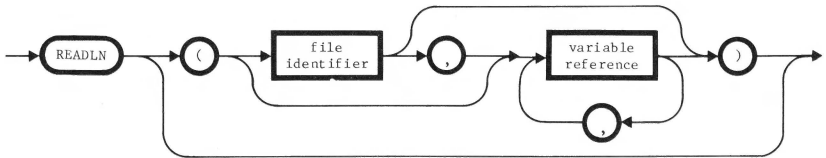
    VAR F: FILE OF REAL;
        I: INTEGER;
        NAME: STRING;

    BEGIN
      {Ask user for pathname.}
      WRITE('Type name of file: ');
      {Accept line typed by user.}
      READLN(NAME);
      {If pathname has no suffix, add customary .DATA suffix.}
      IF POS('.', NAME)=∅ THEN NAME:=CONCAT(NAME, '.DATA');
      {Open with REWRITE since this is a new file}
      REWRITE(F, NAME);
      ...
```

Another useful example is given below under WRITE and WRITELN.

The WRITE Procedure

This procedure may be used only on character files. It allows
characters, strings, and numeric values to be written to a file
as text strings, without the need for explicit use of PUT or
explicit reference to the window variable. The syntax for
calling WRITE is

where the file identifier must refer to an open character file.
If the file identifier is omitted, OUTPUT is assumed. Each
"value specifier" has the following syntax:

The "value expression" may have an INTEGER, REAL, STRING, CHAR,
or LONG INTEGER value; these are values to be written out as text
strings. Also, the identifier of a one-dimensional PACKED ARRAY
OF CHAR may be used; all of the characters in the array are
written out as a string.

A "width expression" must have a positive INTEGER value. If
present, it specifies the minimum number of character positions
to be occupied by the string that is written out. If no width is
specified for a value, the default width is exactly enough
character positions to hold the value. If the value written out
is shorter than the width, spaces are added on the left to fill
the width (i.e., the value is right-justified within the width).

When an INTEGER value is written out, the sign is written only if
it is negative. If the sign is positive, a space is written
instead.

When a REAL value is written out, the rules are slightly
different. The value is always preceded by at least one space,
regardless of the width specification. After the space, the sign
is written only if it is negative. Nothing is written for a
positive sign.

A "decimal places expression" is only allowed after a REAL
value. It must have a positive INTEGER value. If present, it
specifies the number of decimal places to be written out. The
number is written in "fixed-point" form, i.e., the exponent
notation is not used. If necessary, the value is rounded (not
truncated) to the specified number of decimal places.

If a string value or PACKED ARRAY OF CHAR is longer than the
specified width, the value or array is truncated on the right to
fit the specified width.

If the "value expression" has a numeric value, the WRITE
procedure always writes the entire value. The "width expression"
is ignored if the numeric value is longer than the specified
width, or if the width is too short to contain the specified
number of decimal places.

If the number of decimal places is not specified, the value is
rounded to 6 significant digits, the decimal point is placed
between the first and second digits, and the exponent notation is
used.

Regardless of the width and decimal places specified, no more
than 9 significant digits are ever written out.

A numeric value written out by WRITE or WRITELN produces
exactly the same number when read back by READ or READLN
if one of the following conditions is true:
   - the value is written as a 6-digit number and
     read as a 6-digit number; or
   - the value is written as a 9-digit number and
     read as a 9-digit number.

Special characters are handled by the WRITE procedure as
follows:

   - Whenever a RETURN character (ASCII 13) is written to a
     character device, a linefeed (ASCII 1Ø) is
     automatically written out immediately after the RETURN

character. This is not done when output is to a
diskette file.

- When a DLE-blank code (ASCII 16 followed by another
  character) is written to a character device, the
  DLE-blank code is "expanded" to a sequence of spaces.
  The DLE-blank coding is explained in Chapter 12.
  DLE-blank codes are not expanded when output is to a
  diskette file; the DLE character and the next character
  are written like any other characters.

- No special handling is provided for the CTRL-C
  character; but note that the console treats this
  character as a screen-control code. In general, do not
  write any control character to the console unless you
  carefully study the "Screen Control Codes" section of
  the Standard Device Drivers Handbook.

Because of the special handling of control characters,
WRITE is not recommended for control-character
communication with a SOS device driver. To control a
device driver by sending it control characters, use
UNITREAD and UNITWRITE as explained in Chapter 12.

The standard system library file, SYSTEM.LIBRARY , contains
a library unit called PASCALIO . WRITE or WRITELN with a
REAL variable cannot be executed unless this unit is
available in a library at start of execution.

The WRITELN Procedure

WRITELN works exactly like WRITE, except that after the last
value has been written a RETURN character (ASCII 13) is written
to end the line. If output is to a character device, the RETURN
character is automatically followed by a linefeed (ASCII 1Ø).
The syntax for calling WRITELN is

The syntax for a "value specifier" is given above under WRITE. WRITELN with no value specifiers skips to the next line in the output.

The following example uses WRITELN to produce formatted output on the screen, namely a table of the ASCII codes for the digits '∅'..'9':

```
PROGRAM ASCIITABLE;
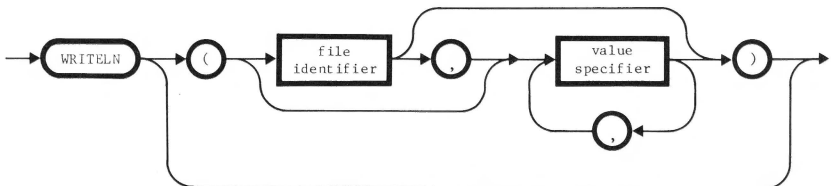
VAR DIGIT: CHAR;

BEGIN
  WRITELN('Character     Code');
  WRITELN;
  FOR DIGIT := '∅' TO '9' DO
    WRITELN(DIGIT:5, ORD(DIGIT):12)
END.
```

The following example program illustrates a number of useful techniques. It uses line-oriented I/O with STRING variables, and performs character manipulations on the STRING variables. It also shows the use of IORESULT to stop the program if the input file does not exist. 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 pathname.}
    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.}
    {$IOCHECK-}
    {Input file should already exist, so open with reset.}
    RESET(INFILE,INNAME);
    {If it doesn't work, complain and stop program.}
    IF IORESULT<>Ø THEN BEGIN
      WRITELN('File not found.');
      EXIT(PROGRAM)
    END;
    {Turn automatic error checking back on.}
    {$IOCHECK+}

    {Get output pathname.}
    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) DO BEGIN
      READLN(INFILE,LINEBUF);
        WHILE POS(PERIOD, LINEBUF)=1 DO DELETE(LINEBUF, 1, 1);
      WRITELN(OUTFILE,LINEBUF)
    END;

    {Now clean up.}
    CLOSE(OUTFILE,LOCK);
    CLOSE(INFILE)
END.
```

The PAGE Procedure

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

    PAGE (FILEID)

where FILEID must be the identifier of an open character file. Note that PAGE(OUTPUT) has the effect of placing the cursor at the top left corner of the screen without clearing the screen.

Additional Details

The following facts about READ and READLN are important if you combine text I/O with GET and PUT calls, or mix reading and writing operations on the same file variable. You may also need to know exactly when EOLN and EOF become true with READLN and with numeric variables.

Note that for mixed reading and writing, the rules given below are more straightforward for INTERACTIVE files than for TEXT files.

After READ with a CHAR variable and an INTERACTIVE file:

- The file buffer variable contains the character that was read, unless EOLN or EOF is true.

- If the next I/O operation is a PUT, WRITE, or WRITELN, the operation begins with the character after the one that was read.

- EOF is true if the character read was beyond the end of the file or was a CTRL-C character from a character device. In this case the value of the file buffer variable is undefined.

- EOLN is true if the character read was the RETURN character. In this case the file buffer variable contains a space.

- EOLN is also true if EOF is true.

After READ with a CHAR variable and a TEXT file:

- The file buffer variable contains the character after the character that was read, unless EOLN or EOF is true.

- If the next I/O operation is a PUT, WRITE, or WRITELN, the operation begins with the second character after the one that was read.

- EOF is true if the character read was the last character in the file or if a CTRL-C was read from a character device. In this case the value of the file buffer variable is undefined.

- EOLN is true if the character read was the last character on the line (not counting the RETURN character). In this case the file buffer variable contains a space.

- EOLN is also true if EOF is true.

After READ with a numeric variable and a TEXT or INTERACTIVE file:

- The file buffer variable contains the character after the last character of the numeric string that was read, unless EOLN or EOF is true.

- If the next I/O operation is a PUT, WRITE, or WRITELN, the operation begins with the second character after the last character of the numeric string.

- EOF is true if the last character of the numeric string was the last character in the file or a CTRL-C was read from a character device. In this case the value of the file buffer variable is undefined.

- EOLN is true if the last character of the numeric string was the last character on the line (not counting the RETURN character). In this case the file buffer variable contains a space.

- EOLN is also true if EOF is true.

After READ with a STRING variable and a TEXT or INTERACTIVE
file:

- The file buffer variable contains a space which
  represents the RETURN character at the end of the line,
  unless EOF is true.

- If the next I/O operation is a PUT, WRITE, or WRITELN,
  the operation begins with the first character on the
  next line.

- EOF is true if the line read was the last line in the
  file. In this case the value of the file buffer
  variable is undefined.

- EOLN is always true.

After READLN with any variable and an INTERACTIVE file:

- The file buffer variable contains a space which
  represents the RETURN character at the end of the line,
  unless EOF is true.

- If the next I/O operation is a PUT, WRITE, or WRITELN,
  the operation begins with the first character on the
  next line.

- EOF is true if the line read was the last line in the
  file. In this case the value of the file buffer
  variable is undefined.

- EOLN is never true.

After READLN with any variable and a TEXT file:

- The file buffer variable contains the first character
  on the next line, unless EOLN or EOF is true.

- If the next I/O operation is a PUT, WRITE, or WRITELN,
  the operation begins with the second character on the
  next line.

- EOF is true if the line read was the last line in the file. In this case the value of the file buffer variable is undefined.

- EOLN is true only when EOF is true.

12

Block File I/O and Device I/O

Introduction

This chapter describes the two lower levels of I/O provided by
Apple III Pascal: block file I/O, and device I/O. These methods
allow higher speed and more direct control of the I/O process, by
omitting most of the automatic features of the higher levels of
I/O.

The last section of this chapter describes the difference between
files of external types "Textfile" and "Asciifile". The
difference is invisible to a Pascal program unless it uses the
methods described in this chapter.

Block File I/O

Block file I/O treats a file variable as a sequence of 512-byte
"blocks"; the bytes are not type-checked but considered as raw
data. This can be useful for applications where the data need
not be interpreted at all during I/O operations.

The external file is still handled as described in earlier
chapters; it has a pathname which is found in a directory, it
must be opened and closed, etc.

To use block file I/O, the file is declared as a block file, and
the BLOCKREAD and BLOCKWRITE functions are used for input and
output.

Block File Declarations

Block files are sometimes called "untyped" files, because they
are declared with no component type. The type for a block file
variable is simply FILE. For example, the declaration

        VAR INFILE, OUTFILE: FILE;

declares two file variables, INFILE and OUTFILE. They have no
component types, so they are block files.

A block file is opened and closed like any other file. The type
of the external file ("Textfile," "Asciifile," "Datafile," etc.)

is totally ignored.

A block file has no buffer variable, and it cannot be used with GET, PUT, or any of the text I/O procedures. It can only be used with RESET, REWRITE, CLOSE, EOF, and the BLOCKREAD and BLOCKWRITE functions described below.

The BLOCKREAD Function

This function is used to transfer one or more 512-byte blocks of data from a block file to a program variable. The value returned is the number of blocks actually transferred. The syntax for calling BLOCKREAD is

where

- The file identifier must be the identifier of an open block file.

- The variable reference refers to the variable into which the blocks of data will be read. The size and type of this variable are not checked; if it is not large enough to hold the data, other program data may be overwritten and the results are unpredictable.

- The "count expression" is an expression of integer value that specifies the number of blocks of data to be transferred. BLOCKREAD will read as many blocks as it can, up to this limit; if the end of the file is reached before the specified number of blocks are read, then EOF will be true and the value returned by

BLOCKREAD indicates how many blocks were actually read.

- The "blocknumber expression" is an expression of integer value that specifies the starting block number in the file (see below).

The blocks in a file are considered to be numbered sequentially starting with $\emptyset$. The system keeps track of the "current" block number in each open block file; this is block $\emptyset$ immediately after the file is opened. Each time a block is read, the current block number is incremented. If the blocknumber expression is omitted in a call to BLOCKREAD, the transfer begins with the current block. Thus the transfers are sequential if the blocknumber expression is never used; if a blocknumber expression is used, it provides random access to blocks.

After BLOCKREAD, EOF is true if the last block in the file was read.

The BLOCKWRITE Function

This function is used to transfer one or more 512-byte blocks of data from a program variable to a block file. The value returned is the number of blocks actually transferred. The syntax for calling BLOCKWRITE is

where

- The file identifier must be the identifier of an open block file.

- The variable reference refers to the variable from
  which the blocks of data will be read. The size and
  type of this variable are not checked; if it is not
  large enough to contain the specified number of blocks
  of data, other program data will be written out to the
  file.

- The "count expression" is an expression of integer
  value that specifies the number of blocks of data to be
  transferred. BLOCKWRITE will write as many blocks as
  it can, up to this limit; if the available space is
  used up before the specified number of blocks are
  written, then EOF will be true and the value returned
  by BLOCKWRITE indicates how many blocks were actually
  written.

- The "blocknumber expression" is an expression of
  integer value that specifies the starting block number
  in the file (see explanation above under BLOCKREAD).

Using Block I/O

The following program is an example of the use of block I/O. It
simply copies one diskette file to another. This could be done
by transferring one block at a time, but the program can execute
faster by transferring two blocks at a time, using a buffer
variable whose size is 1Ø24 bytes. Since a file might contain an
odd number of blocks, the program must make sure that after each
BLOCKREAD, the subsequent BLOCKWRITE transfers the same number of
blocks that were read.

```
PROGRAM FILECOPY;
{Copy one diskette file to another.}

VAR INFILE, OUTFILE: FILE;

PROCEDURE OPENINPUT;
{Open the input file.}
  VAR INNAME:STRING;
  BEGIN
    WRITE('Type pathname of input file: ');
    READLN(INNAME);
    RESET(INFILE, INNAME)
  END;
```

```
PROCEDURE OPENOUTPUT;
{Open the output file.}
  VAR OUTNAME:STRING;
  BEGIN
    WRITE('Type pathname of output file: ');
    READLN(OUTNAME);
    REWRITE(OUTFILE, OUTNAME)
  END;

PROCEDURE TRANSFER;
{Transfer blocks two at a time from input file, to buffer
 variable, to output file, until end of input file.}

  VAR COUNT: INTEGER;
      BUF: PACKED ARRAY[1..1024] OF CHAR;
      {The important thing about BUF is that it is exactly
       1024 bytes in size--equivalent to two blocks.  The
       BLOCKREAD and BLOCKWRITE functions do not check for
       this, and they do not care that it happens to be a
       packed array of char.}

  BEGIN
    WHILE NOT EOF(INFILE) DO BEGIN
      COUNT := BLOCKREAD(INFILE, BUF, 2);
      COUNT := BLOCKWRITE(OUTFILE, BUF, COUNT)
    END
  END;

BEGIN {Main program}
  OPENINPUT;
  OPENOUTPUT;
  TRANSFER;
  CLOSE(INFILE);
  {LOCK to make new file permanent.}
  CLOSE(OUTFILE, LOCK)
END.
```

Device I/O

The Pascal system identifies each peripheral device by a unit
number and a unit name. SOS device names may also be used. The
numbers and names for standard devices are

SOS DEVICE NAME	PASCAL UNIT #	PASCAL UNIT NAME
.CONSOLE	1	CONSOLE:
.CONSOLE	2	SYSTERM:
.GRAFIX	3	GRAPHIC:
.D1	4	(volume name)
.D2	5	(volume name)
.PRINTER	6	PRINTER:
.RS232	7	REMIN:
.RS232	8	REMOUT:
.D3	9	(volume name)
.D4	1∅	(volume name)

The distinction between units 1 and 2 (CONSOLE: and SYSTERM:) is that I/O operations using SYSTERM: (unit 2) do not cause typed characters to be echoed on the screen. The built-in Pascal file identifier KEYBOARD is associated with the SYSTERM: unit, and the built-in Pascal file identifiers INPUT and OUTPUT are associated with the CONSOLE: unit.

The distinction between units 7 and 8 (REMIN: and REMOUT:) is that unit 7 is used for input and unit 8 is used for output. Both are associated with the SOS device .RS232.

Note that if there is a printer-like driver present (e.g., .SILENTYPE) and no .PRINTER , the printer-like driver is assigned the Pascal unit name PRINTER: and its unit number is 6.

Non-standard devices are assigned sequential unit numbers in the range 128 through 255 according to their order in SOS.DRIVER. (See Standard Device Drivers Handbook.)

The procedures described below can be used for direct communication with any peripheral device.

The UNITREAD and UNITWRITE Procedures

These are the procedures that perform device-oriented I/O. They are dangerous procedures. Unlike the other I/O procedures, these do not offer any protection against mistakes. In particular, UNITWRITE allows you to write any block on a disk without knowing what it contains; thus you can easily destroy a disk directory. Use these procedures cautiously. The syntax is

where

The "unitnum expression" is an expression with an
integer value, which is the unit number of an I/O
device.

The variable reference refers to a program variable,
whose contents are to be transferred to or from the
unit. As with the block I/O procedures (see above),
the type and size of this variable are not checked; if
the size of the variable does not match the specified
number of bytes to be transferred, results are
unpredictable.

The "length expression" is an expression with an
integer value, which specifies the number of bytes to
transfer.

The "blocknum expression" is an expression with an
integer value. It is meaningful only when using a disk
drive and is the absolute block number at which the
transfer will start. If the blocknum expression is
omitted and the unit is a disk drive, the transfer will
start at block Ø.

The "mode expression" is an expression with an integer
value; if it is omitted, the default is Ø. It controls
options which are described below.

For UNITWRITE, the options controlled by the "mode" parameter
apply only to character devices; they do not apply to
block-structured devices. Both options are enabled by default,
if no "mode" parameter is supplied. They are designed to handle
the special coding found in "Textfiles", and are convenient when
text is read from such a file with UNITREAD and then output via
UNITWRITE to a character device.

One option is conversion of DLE-blank codes. These are found in
"Textfiles", as explained in the last section of this chapter.
On output to a character device, this option detects the
DLE-blank code and converts it into a sequence of spaces.

Conversion of DLE-blank codes is disabled by a "mode" value that
has a one in Bit 2 (see below).

The other option is automatic linefeeds. In character files, the
end of each line is marked by a RETURN character (ASCII 13)
without any linefeed character. On output to a character device,
the automatic linefeed option inserts an LF character (ASCII 1Ø)
after every RETURN.

Automatic linefeeds are disabled by a "mode" value that has a one
in Bit 3 (see below).

For UNITREAD, the only option is CTRL-C recognition (enabled by
default). When the CTRL-C character (ASCII 3) is recognized by
UNITREAD, the effect is to terminate the input. Any bytes in the
destination variable that remain unused at this point are filled
with Ø's.

When unit 1 (CONSOLE:) is read with UNITREAD, characters
typed on the keyboard are echoed (as with other methods of
input). The echoing is performed by UNITWRITE, using the
same mode parameter supplied to UNITREAD. Usually this
has no noticeable effect, but it might cause a surprising
result with certain modes and certain input characters.

Only Bit 2 and Bit 3 of the "mode" value have any significance.
Bits Ø, 1, and 4..12 are reserved for future use; bits 13..15 are
unassigned.

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) enables all options.

- Mode = 4 disables DLE conversion and CTRL-C
  recognition, and enables automatic linefeeds.

- Mode = 8 disables automatic linefeeds and enables DLE
  conversion and CTRL-C recognition.

- Mode = 12 disables all options. This mode should be
  used for control-character communication with SOS
  driver programs.

Note that UNITREAD and UNITWRITE are not controlled in any
way by the format in which files are stored on a diskette;
they treat an entire diskette as one sequence of blocks.
Therefore, files on Apple II formatted diskettes and files
on SOS-formatted diskettes will not be accessed in the
same way by these built-ins. Don't use these procedures
with diskette files unless you know exactly what you're
doing.

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 (UNITNUM)

where UNITNUM is an expression with an integer value, which is
the unit number of an I/O device.

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 unit is
present in the system). Note that UNITCLEAR (1) flushes the
type-ahead buffer for the console and resets certain keyboard and
screen parameters.

For any device except the console, UNITCLEAR is equivalent
to the SOS device control call with the control parameter
set to zero to indicate a RESET operation.

The UNITSTATUS Procedure

This procedure communicates with the SOS device driver program associated with a specified unit. It can either check the current status parameters for the device's input or output channel, or perform a control operation on the device.

Reserved	Driver dependent code	S/C	O/I

15   13 12   2   1   Ø

UNITSTATUS: OPTION Format

The form is

    UNITSTATUS (UNITNUM, DATA, OPTION)

where

    UNITNUM is an expression with an integer value that is the unit number of an I/O device.

    DATA is a variable reference. The variable is assumed to contain either control information to be transferred to the device driver, or status information to be retrieved from it. For most devices, this variable should be an ARRAY[Ø..29] OF INTEGER, but this may differ for certain devices. The type and size of this variable are not checked.

    OPTION is an expression with an integer value whose individual bits control the operation of the UNITSTATUS procedure as follows:

        Bit Ø: If this bit is Ø, the output channel of the unit is specified; if it is 1, the input channel is specified. For most devices, there is no distinction between input and output channels and this bit is ignored; it is also ignored if the device has

only one channel (e.g., a printer).

Bit 1:   If this bit is Ø, UNITSTATUS will put
the current status parameters of the device
driver into the specified program variable
(calling the SOS device status procedure).
If it is 1, UNITSTATUS will use the values in
the program variable to update the status
parameters of the device driver (using the
SOS device control procedure).

Bits 2..12:   These bits are sent to the
device driver, which uses them to select a
particular status or control operation.   For
further information on these status codes and
control codes, see the Standard Device
Drivers Handbook.

Bits 13..15:   These bits are reserved for
future use.   They should be set to zero.

As an example of the use of UNITSTATUS, consider the problem of
determining whether a character is present in the keyboard
typeahead buffer.   The console driver will return the number of
characters in the typeahead buffer in response to a device status
call with a parameter of 5.   UNITSTATUS, with the following
parameters, will issue this device status call:

UNITNUM = 1 to specify the console unit.

A reference to an INTEGER variable to contain the
result returned from the console driver.   (Note that in
this case we do not use an ARRAY[Ø..29] OF INTEGER; the
type of variable used here depends on the device.)

OPTION = 21, which has the following bit values:

Bit Ø = 1 to specify the input channel of the
console.

Bit 1 = Ø to specify a device status call
(retrieve status information from device
driver).

Bits 2..12 cause the console driver to return
the number of characters in the typeahead

buffer. Bits 2 and 4 are set to 1; all
others in the range are set to zero.

Reserved	Driver dependent code	S/C	O/I
Ø Ø Ø	Ø Ø Ø Ø Ø Ø Ø Ø 1 Ø 1	Ø	1
15 13 12	2	1	Ø

UNITSTATUS: OPTION = 21

The following function uses this UNITSTATUS call. It returns
true if the typeahead buffer contains one or more characters;
thus it can be used to find out whether any character has been
typed since the last time the keyboard was read or cleared.

```
FUNCTION KEYPRESS: BOOLEAN;
  VAR CHARCOUNT: INTEGER;
  BEGIN
    CHARCOUNT:= Ø;
    UNITSTATUS(1, CHARCOUNT, 21);
    KEYPRESS := CHARCOUNT <> Ø
  END;
```

Note that this KEYPRESS function is provided in the APPLESTUFF
library unit (see Appendix D).

Textfiles and Asciifiles

When a diskette file is created, it is assigned one of the
external file types described in the Introduction, Filer, and
Editor manual. Character files are generally of external type
"Asciifile" or "Textfile"; but note that for purposes of Pascal
program I/O, the other external types ("Datafile" in particular)
are equivalent to "Asciifile".

When you create a file via the Filer or via a user Pascal
program, the type depends on the suffix used as the last part of
the pathname. Files created with the suffix .ASCI are of type

"Asciifile". Files created with the suffix .TEXT are of type
"Textfile".

The Editor creates "Textfiles" by default, and can create
"Asciifiles" on request. The Editor reads files as "Textfiles"
unless they are of external type "Asciifile", in which case it
assumes no special formatting.

Textfile Structure

"Textfiles" have a special physical format that includes other
data besides the characters that appear to a Pascal program.
When you access a "Textfile" with the I/O procedures of Chapters
10 and 11, all of this extra data is invisible; but when you use
block I/O or device I/O, the extra data and the special
formatting are visible.

The special format of a "Textfile" is as follows:

- There are two blocks (1024 bytes) of "header"
  information at the beginning of the file. When RESET
  opens a "Textfile" (using a file variable of type TEXT
  or INTERACTIVE), these 1024 bytes are skipped. When
  REWRITE opens a "Textfile" (using a variable of type
  TEXT or INTERACTIVE), these 1024 bytes are
  automatically written out with default values. In
  other words, the header blocks are normally invisible
  to a Pascal program. However, if the file variable is
  not a character file, the header information is treated
  as ordinary data.

- The rest of the file consists of two-block "pages."
  Each page contains lines of text, separated from each
  other by RETURN characters (ASCII 13). No line ever
  crosses a page boundary; thus a page contains only
  whole lines. After the last line on a page, the
  remainder of the page is filled with NUL characters
  (ASCII 00). READ and READLN skip the NUL characters,
  and WRITE and WRITELN provide them automatically. Thus
  this "page" formatting is also normally invisible to a
  Pascal program. However, if the file variable is not a
  character file, the NUL characters are treated as
  ordinary data.

- A sequence of leading spaces in a line may be
  compressed to a "DLE-blank code." This is a DLE

control character (ASCII 16) followed by one byte
containing the number of spaces to indent plus 32
(decimal). This saves a considerable amount of space
in files where indentation is used heavily. The Editor
is the main creator of DLE-blank codes; it usually
outputs a DLE-blank code where a sequence of spaces
occurs at the beginning of a line. Of course, Pascal
user programs may also use this coding in creating
"Textfiles".

GET, READ, and READLN convert DLE-blank coding to
actual spaces on input from a "Textfile" to a file
variable of type TEXT or INTERACTIVE; thus the
compression of spaces is also normally invisible to a
Pascal program.

Note that it is still possible for a "Textfile" to
contain actual sequences of spaces, including leading
spaces on a line, depending on how the data was
actually created and output. Also, a line with no
indentation may or may not be preceded by a DLE
character and an indent code value of 32 (meaning Ø
indentation).

Various parts of the system that deal with files of characters
(such as the Editor and the Compiler) are designed to take
advantage of the special "Textfile" format. For most purposes,
it is recommended that you use the "Textfile" type for any
character files created by your programs.

Asciifile Structure

Text files created by BASIC are "Asciifiles"; since the Pascal
Editor can handle these files, it can be used to edit BASIC text
files. (See the Introduction, Filer, and Editor manual.)
Likewise, Pascal programs can be written to process such files.

The "Asciifile" type can also be used for files created by Pascal
programs, in certain situations--for example, where
character-level random access is required.

Note that for purposes of Pascal program I/O, there is actually
no distinction between "Asciifiles" and other diskette files that
are not "Textfiles".

An "Asciifile" contains a sequence of characters; for purposes of a Pascal program, each character that is in the external file appears as one record of a file variable of type TEXT or INTERACTIVE.

Physically, the file is a sequence of 512-byte blocks (not 2-block pages). There are no header blocks; the first character in the file is the first byte in the first block.

Lines in an "Asciifile" may cross block boundaries; there is no filling with NUL characters, except in the last block where any unused bytes after the last file character are filled with NULs.

On output to an "Asciifile" from the Editor, sequences of spaces are not converted to DLE-blank codes. If an "Asciifile" does contain a DLE-blank code, it is not interpreted or converted on input to a Pascal program; it simply appears as two characters, namely the DLE character followed by some character whose ASCII code is the indent code. Thus all spaces in an "Asciifile" should be represented by actual space characters.

13

Special-Purpose Built-Ins

Introduction

This chapter describes two sets of built-in features of Apple III
Pascal: the byte-oriented features and three "miscellaneous"
procedures.

Byte-Oriented Features

These features allow a program to treat a program variable as a
sequence of bytes, without regard to data types. The SIZEOF
function can be used to determine the number of bytes in a
variable; this is important since none of the other features
described here do any checking on variable sizes.

These features make it very easy to overwrite memory areas
unintentionally; use them with caution.

Typically, the byte-oriented features are used to access packed
character arrays or other packed variables whose elements are
stored as byte values. However, they are not restricted to such
uses.

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 in connection with
the FILLCHAR, MOVERIGHT, and MOVELEFT built-ins (see below). The
form is

     SIZEOF (IDENTIFIER)

where IDENTIFIER is either a type identifier or a variable
identifier. It must not be BYTESTREAM, WORDSTREAM, a file type,
or the identifier of a variable of any of these types, because
these types do not have a definite size. (The BYTESTREAM and
WORDSTREAM types are described further on in this chapter.)

For example, if AREA is declared as

    AREA: PACKED ARRAY[∅..511] OF ∅..255;

(a packed array of 512 byte-size integer values), then the
function reference

    SIZEOF(AREA)

will return the value 512.

The FILLCHAR Procedure

This procedure fills a specified range of memory bytes with the
ASCII code of a specified character. The form is

    FILLCHAR (DEST, COUNT, CHARACTER)

where

    DEST is a variable reference and may refer to a
    variable of any type except a file type. The first
    byte of this variable is the beginning of the range of
    bytes to be filled.

    COUNT is an expression with an integer value which
    specifies the number of bytes to be filled.

    CHARACTER is an expression of any scalar type whose
    ordinal value MOD 256 is copied into each byte in the
    specified range.

For example, if AREA is declared as in the previous example, then
each of the 512 bytes of AREA can be filled with the numerical
value 65 (the ASCII code for 'A') by either of the following
statements:

    FILLCHAR(AREA, SIZEOF(AREA), 'A')

        or

    FILLCHAR(AREA, SIZEOF(AREA), CHR(65))

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 one of

        SCAN (LIMIT, =CHARACTER, SOURCE)   or
        SCAN (LIMIT, <>CHARACTER, SOURCE)

where

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

        CHARACTER is an expression with a CHAR value. If it is
        preceded by the = symbol, then SCAN searches for the
        specified character. If it is preceded by the <>
        symbol, then SCAN searches for any character that is
        not the specified character.

        SOURCE is a variable reference that may refer to 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 the number of bytes specified by LIMIT. 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 be negative or zero.

Examples: Suppose that DEM is declared as follows:

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

and then the first 50 elements of DEM are loaded with the characters

        .....THE NEXT FIVE NOTES OF THE SCALE IN SEQUENCE.

We then have the following:

SCAN(-26,=':',DEM[3∅])   will return -26

SCAN(1∅∅,<>'.',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, DEST, COUNT)

MOVERIGHT (SOURCE, DEST, COUNT)

where

SOURCE is a variable reference that may refer to a variable of any type except a file type. The first byte of this variable (lowest address) is the beginning of the range of bytes whose values are copied.

DEST is a variable reference that may refer to a variable of any type except a file type. The first byte of this variable (lowest address) is the beginning of the range of bytes that the values are copied into.

COUNT is an expression with an integer value, which specifies the number of bytes to be moved.

MOVELEFT starts from the "left" end of the source range (lowest address). It proceeds to the "right" (higher addresses), copying bytes into the destination range, starting at the lowest address of the destination range.

MOVERIGHT starts from the "right" end of the source range (highest address). It proceeds to the "left" (lower addresses), copying bytes into the destination range, starting at the highest address 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 particular this consideration applies when source and destination are subarrays of the same array. 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 BYTESTREAM and WORDSTREAM Types

These special data types allow a procedure or function to accept an array of indefinite length as a VAR parameter. Within the procedure or function, the array can be indexed with any non-negative INTEGER value. BYTESTREAM is used for a PACKED ARRAY OF CHAR or STRING, and WORDSTREAM is used for an array of scalar type.

BYTESTREAM and WORDSTREAM act within a procedure as if they were declared as follows:

```
TYPE BYTESTREAM = PACKED ARRAY[∅..?] OF CHAR;
     WORDSTREAM = ARRAY[∅..?] OF INTEGER;
```

where the "?" means that the upper bound of the index type is not defined. BYTESTREAM is compatible with any PACKED ARRAY of byte-sized quantities such as CHAR, or any STRING type. WORDSTREAM is compatible with any non-packed array of scalars. The actual parameter may be indexed.

BYTESTREAM and WORDSTREAM can occur only in the parameter list of a procedure or function declaration.

For example, suppose that your program contains the declarations

```
VAR A : ARRAY[∅..2∅] OF INTEGER; {21 ELEMENTS}
    B : ARRAY[1..512] OF INTEGER; {512 ELEMENTS}
```

Now suppose that you want to write a procedure that will accept, as a VAR parameter, any array of INTEGER. The procedure also accepts the number of elements in the array and an INTEGER value;

it fills all elements of the array with the INTEGER value.

```
PROCEDURE FILL (VAR INTARRAY:WORDSTREAM; COUNT,N:INTEGER);
   VAR I:INTEGER;
   BEGIN
     FOR I:=∅ TO COUNT-1 DO INTARRAY[I]:=N
   END;
```

Now to fill array A with the value 5 and array B with the value ∅, write

```
FILL(A, 21, 5);
FILL(B, 512, ∅)
```

Similarly, the following example shows how the type BYTESTREAM can be used in a procedure that accepts any PACKED ARRAY OF CHAR as a VAR parameter. The procedure goes through this array and replaces each lower-case letter that it finds with the corresponding upper-case letter. As in the previous example, the number of elements in the array must also be supplied as a parameter.

```
PROCEDURE UPPERCASE (VAR CHARRAY:BYTESTREAM; COUNT:INTEGER);
   VAR OFFSET, I:INTEGER;
   BEGIN
     OFFSET:=ORD('a') - ORD('A');
     FOR I:=∅ TO COUNT-1 DO
       IF CHARRAY[I] IN ['a'..'z'] THEN
         CHARRAY[I]:=CHR(ORD(CHARRAY[I]) - OFFSET)
   END;
```

Within a procedure or function the type must be treated as a PACKED ARRAY OF CHAR (for type BYTESTREAM) or an ARRAY OF INTEGER (for type WORDSTREAM). Range checking is never done on a parameter of type BYTESTREAM or WORDSTREAM.

Also note that within a procedure that has a BYTESTREAM parameter, you cannot use the BYTESTREAM as a parameter to WRITE or WRITELN. However you can use an indexed element of the BYTESTREAM as a parameter to WRITE or WRITELN, since each element of a BYTESTREAM is just a CHAR variable.

Miscellaneous Procedures

The PWROFTEN Function

This fast machine-code function returns a real value which is 1∅
to a specified (integer) power. The form is

    PWROFTEN (EXPONENT)

where EXPONENT is an expression with an integer value in the
range ∅..37. PWROFTEN returns the value of 1∅ to the EXPONENT
power.

The GOTOXY Procedure

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

    GOTOXY (XCOORD, YCOORD)

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

The TIME Procedure

The form for calling TIME is

    TIME (HITIME, LOTIME)

where HITIME and LOTIME are references to integer variables; TIME
sets these variables to zero. This function is available only
for compatibility with Apple II.

To access the Apple III system's internal date and time,
use procedures in the APPLESTUFF library unit. (See
Appendix D.)

The TREESEARCH Function

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

    TREESEARCH (ROOTPTR, NODEPTR, NAMEID)

where

> ROOTPTR is a pointer to the root node of the tree to be searched.
>
> NODEPTR is a reference to a pointer variable to be updated by TREESEARCH.
>
> NAMEID 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 type

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

                ...{other fields can be anything}...

        END;

The actual identifier of the type and the field identifiers are not important; TREESEARCH assumes only 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:

> ∅: The name passed to TREESEARCH (as the third
> parameter) has been found in the tree. The node
> pointer (second parameter) 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 the node pointer.
>
> −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 the node pointer.

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

The IDSEARCH Procedure

IDSEARCH is a very fast machine-language routine that scans Apple
III Pascal source language for identifiers and identifies the
reserved words. The Compiler uses IDSEARCH, and it is also
available to programmers whose programs read Pascal source
language. Note that the procedure scans only identifiers—you
will have to scan special characters and comments yourself. The
syntax is

    IDSEARCH (OFFSET, BUFFER)

To use IDSEARCH, you must include the following declarations in
your program. Note that the variables (except for BUFFER) must
be declared in exactly the order and types shown.

```
     TYPE
       {SYMBOL is the enumerated type of symbols in the Pascal
          language, the lines in comments are the identifiers
          corresponding to the symbols }

SYMBOL =
          {                                          DO    TO  }
          (ident,comma,colon,semicolon,lparent,rparent,dosy,tosy,
          {DOWNTO    END    UNTIL    OF    THEN    ELSE        }
           downtosy,endsy,untilsy,ofsy,thensy,elsesy,becomes,
          {                              BEGIN    IF    CASE   }
           lbrack,rbrack,arrow,period,beginsy,ifsy,casesy,
          {REPEAT    WHILE    FOR    WITH    GOTO    LABEL    CONST  }
           repeatsy,whilesy,forsy,withsy,gotosy,labelsy,constsy,
          {TYPE    VAR    PROCEDURE FUNCTION   **      FORWARD }
           typesy,varsy,procsy,   funcsy,   progsy,forwardsy,
          {                              NOT    ***    OR      }
           intconst,realconst,stringconst,notsy,mulop,addop,
          {IN     SET    PACKED    ARRAY    RECORD    FILE      }
           relop,setsy,packedsy,arraysy,recordsy,filesy,othersy,
          {           USE    UNIT    INTERFACE IMPLEMENTATION   }
           longconst,usessy,unitsy,intersy,   implesy,
          {EXTERNAL    OTHERWISE                                }
           externlsy,otherwsy);

          {**: progsy is returned on both PROGRAM and SEGMENT}
          {***: mulop is returned on AND, DIV, MOD            }

          {OPERATOR expands the multiplicative, additive and
               relational operators (mulop, addop and relop)}

     OPERATOR = {          AND    DIV MOD              OR       }
                 (MUL,RDIV,ANDOP,IDIV,IMOD,PLUS,MINUS,OROP,LTOP,
                 {                              IN               }
                  LEOP,GEOP,GTOP,NEOP,EQOP,INOP,NOOP);

     ALPHA = PACKED ARRAY [1..8] OF CHAR;
```

```
VAR
 {the next four variable must be declared in order shown}
 OFFSET: INTEGER;
 SY: SYMBOL;
 OP: OPERATOR;
 ID: ALPHA;

 BUFFER: PACKED ARRAY [Ø..1Ø23] OF CHAR;

 ...
   IF BUFFER[OFFSET] in ['A'..'Z','a'..'z'] THEN BEGIN
     IDSEARCH(OFFSET, BUFFER);
   END;
 ...
```

IDSEARCH begins looking for an identifier at the character
pointed to by BUFFER[OFFSET] and assumes that this character is
alphabetic. IDSEARCH produces the following results:

- BUFFER[OFFSET] points to the character following the
  identifier just found.

- ID contains the first 8 alphanumeric characters of the
  identifer just found, left-justified and padded with
  spaces as necessary.

- SY contains the symbol associated with the identifier
  just found if the identifier is a reserved word, or
  IDENT if the identifier is not a reserved word. E.g.,
  the identifier 'MOD' translates to MULOP; the
  identifier 'ARRAY' translates to ARRAYSY; and the
  identifier 'MYLABEL' translates to IDENT.

- OP contains the operator value which corresponds to the
  identifier just found if the identifier is an operator,
  or NOOP if the identifier is not an operator. E.g.,
  the identifier 'MOD' translates to IMOD; the
  identifier 'ARRAY' translates to NOOP; and the
  identifier 'MYLABEL' translates to NOOP.

An example of a procedure which uses IDSEARCH follows:

```
BEGIN
  ...
  IF BUFFER[OFFSET] in ['A'..'Z','a'..'z'] THEN BEGIN
    IDSEARCH(OFFSET, BUFFER);
  END;
  ...
END.
```

An algorithmic representation of IDSEARCH follows:

```
PROCEDURE IDSEARCH ( VAR OFFSET: INTEGER; VAR BUFFER: BYTESTREAM);

  {ScanIdentifier increments OFFSET until BUFFER[OFFSET] is
   no longer part of an identifier, copying the first 8
   alphanumeric characters passed into ID (right justified,
   padding with spaces)

   LookUpReservedWord translates an identifier into the
       associated symbol (defaulting to IDENT)
   LookUpOperator translates an identifier into the
       associated operator (defaulting to NOOP)
  }

  BEGIN
    ID := ScanIdentifier(OFFSET, BUFFER);
    SY := LookUpReservedWord(ID);
    OP := LookUpOperator(ID);
  END;
```

14

Library Units

Introduction

So far we have seen Pascal programs, which are compiled into codefiles that can be executed directly via the RUN and XECUTE commands. Now we will consider units, which are compiled into libraries.

Units are collections of procedures that are separately compiled and then invoked as modular components of a host program. A compiled unit contains code for "public" procedures, functions, types, constants, and variables. These are available to any host program that uses the unit, just as if they were defined in the host program itself.

Units provide many advantages to the Pascal programmer.

- Units allow programs to be partitioned into logical chunks.

- Units allow programs to be written in small, separately compilable sections which are faster to compile than a single large program and which require less memory at compile time.

- Individual units can be modified without recompiling the main (host) program.

- Units contain groups of common procedures and functions that can be used in more than one program. For instance, the APPLESTUFF unit is available to all Pascal programs.

- Units are a more flexible form of program segmentation than SEGMENT procedures because you can have more than one procedure in a unit.

- Units give the Pascal programmer controlled access to "private" data structures.

Units reside in libraries. A library is a special kind of codefile that is not directly executed; instead, a compiled unit contained in a library can be used by one or more programs. To use a unit, a program must contain a USES declaration with the

name of the unit; the program is then called a host program.

Two or more libraries can be combined into one, and units can be moved from one library to another, by means of the LIBRARY utility program described in the <u>Program Preparation Tools</u> manual. Thus one library file can contain a number of different compiled units.

For example, the Pascal system comes with a library called SYSTEM.LIBRARY which contains code for several units (see Appendices A through E); one of the units is called APPLESTUFF, and it provides a set of procedures and functions for using special features of the Apple III. To use these procedures and functions, a program need only have the declaration

    USES APPLESTUFF;

after the program heading. The program can then call APPLESTUFF procedures such as SOUND and JOYSTICK.

This chapter also explains how to create and compile your <u>own</u> units. The output from compiling a unit is a codefile that can be used as a library; two or more of these files can be combined into a single library by means of the LIBRARY utility.

Regular Units

There are two kinds of units called "regular" and "intrinsic." 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. The diagram at the end of this section illustrates how regular units are created and used.

By default, regular units are assumed to be in the SYSTEM.LIBRARY file on the system diskette; however, you can use a regular unit that is in another library. In this case, you must use the USING option of the compiler to tell the compiler which library file contains the unit. See Appendix F.

If a regular unit used by your program is contained in the SYSTEM.LIBRARY file, a Run command will automatically invoke the Linker to do the necessary linking. Otherwise, you must use the

Compile command to compile the program and then the Link command
to explicitly invoke the Linker.

regular units diagram

Intrinsic Units

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. The Linker is not
needed; an intrinsic unit is "prelinked." The diagram at the end

of this section illustrates how intrinsic units are created and
used.

Intrinsic units help to reduce the size of the host program's
codefile. Also, an intrinsic unit can be modified and recompiled
without the need to relink.

Note that an intrinsic unit <u>must</u> be either in the program library
(see below) or in the SYSTEM.LIBRARY file on the system
diskette. The library that contains the unit must be on line
both when the program is compiled and when the program is
executed.

The Compiler's NOLOAD and RESIDENT options (see Chapter 15
and Appendix F) allow further control over the handling of
intrinsic units.

intrinsic units diagram

Writing a Unit

One or more units can be written in a single file and compiled together; the syntax for a "compilation" (i.e., something that can be compiled) is

compilation

Note that a period is required at the end of every compilation (since a program ends with a period).

The source text for a unit has a form somewhat similar to a Pascal program. The overall syntax of a unit is

unit

The syntax for a unit heading depends on whether the unit is regular or intrinsic, as explained in the Interface section of this chapter.

The INTERFACE declares the parts of the unit that are
"public"--that is, the types, constants, variables, procedures,
and functions that can be referenced by a host program. For
procedures and functions, only the headings are given in the
INTERFACE. The syntax is shown below.

The IMPLEMENTATION is omitted if the INTERFACE does not declare
any public procedures or functions. If present, the
IMPLEMENTATION declares the parts of the unit that are
"private"--that is, the labels, constants, variables, procedures,
and functions that are used within the unit but cannot be
referenced by a host program. Also, the public procedures and
functions whose headings appear in the INTERFACE are redeclared
here with their bodies included. The syntax is shown below.

If a compound statement appears at the end, it is called an
"initialization." This is the "main program" of the unit, and is
automatically executed at the beginning of the host program. If
the initialization is omitted, the unit is terminated with a
single END.

Like a program, a unit is terminated with a period.

Writing Regular Units
The heading of a regular unit has the syntax

regular unit heading

where the identifier is the identifier of the unit to be
referenced in the USES declaration of a host program.

The compiled 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.

Writing Intrinsic Units

The heading of an intrinsic unit has the syntax

intrinsic unit heading

where the csegnum and dsegnum constants are the segment numbers
to be used by the unit at run time. By definition, segment
numbers range from 0 to 63, but certain of these numbers are
reserved for the system (see below).

The unit will have a data segment if it declares any variables
not contained in procedures or functions in either the INTERFACE
or IMPLEMENTATION. In this case the word DATA and a data segment
number are required; otherwise they must be omitted.

The code segment will have segment number csegnum and the data
segment (if there is one) will have segment number 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 Program Preparation Tools manual). In
choosing segment numbers for an intrinsic unit, there are two
constraints:

- No two units in the same library should have the same
  segment number. If they do, the system may use the
  wrong unit.

- When the host program runs, the various segment numbers
  used by the program and the system must not conflict.

Observe the following:

- While any program is executing, the main program body
  uses segment 1.   Therefore, never use this segment
  number for an intrinsic unit.

- Segments Ø, 2 through 6, and 59 through 63 are reserved
  for use by the system.

- The standard library units LONGINTIO and PASCALIO are
  permanently assigned segment numbers 3Ø and 31,
  respectively. Segment 3Ø will be loaded if you use the
  built-in STR function or any long integer operation.
  Segment 31 will be loaded if you use the built-in SEEK
  function, or any of the following with a real
  argument: READ, READLN, WRITE, or WRITELN.

- If the program declares any SEGMENT procedures or
  functions or uses any regular units, these procedures,
  functions, and regular units use sequentially
  increasing segment numbers starting at 7.

- Each unit used by the program uses the segment number
  shown in the library listing. To see what these
  numbers are, use the LIBMAP utility as described in the
  Program Preparation Tools manual.

Generally, it is a good idea to use segment numbers in the range
from 16 through 58, excluding those that are already used in the
library.

For a complete explanation of the meaning of segment
numbers, see the next chapter.

The INTERFACE

The INTERFACE immediately follows the unit's heading. It
contains declarations of 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.

When a procedure or function is declared in the INTERFACE, only
the procedure or function heading is given. The rest of the
procedure or function is declared in the IMPLEMENTATION.

No SEGMENT procedures or functions are allowed in a unit.

Include files are not allowed within the INTERFACE
section.

If the unit uses another unit (as discussed further on in this
chapter), a USES declaration is placed at the beginning of the
INTERFACE. Thus the syntax of the INTERFACE is

interface

Note that no label declarations are allowed in the INTERFACE.

The IMPLEMENTATION

The IMPLEMENTATION, if present, immediately follows the last declaration in the INTERFACE. It must be omitted if the INTERFACE does not declare any procedures or functions.

The IMPLEMENTATION begins by declaring those labels, constants, and variables 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.
Other private procedures and functions may be declared here as
necessary.

The syntax of an IMPLEMENTATION is

implementation

Note that no USES declaration or type declarations are allowed in
the IMPLEMENTATION.

Also, no file variable may be declared anywhere in the IMPLEMENTATION, except as a VAR parameter of a procedure or function.

The Initialization

At the end of the IMPLEMENTATION, following the last function or procedure, an "initialization" may be written. This is a compound statement followed by a period, like the body of a program. The resulting code runs automatically when the host program is executed, before the host program's own code is executed. It can be used to make any preparations that may be needed before the procedures and functions of the unit can be used. The example below shows this. If no initialization is used, the unit must still end with the word END followed by a period. The initialization code is executed exactly once.

An Example Unit

The following is a complete intrinsic unit that has a data segment and an initialization, to demonstrate the information given above. It provides convenient features for opening textfiles.

```
UNIT OPENS;
INTRINSIC CODE 17 DATA 18;
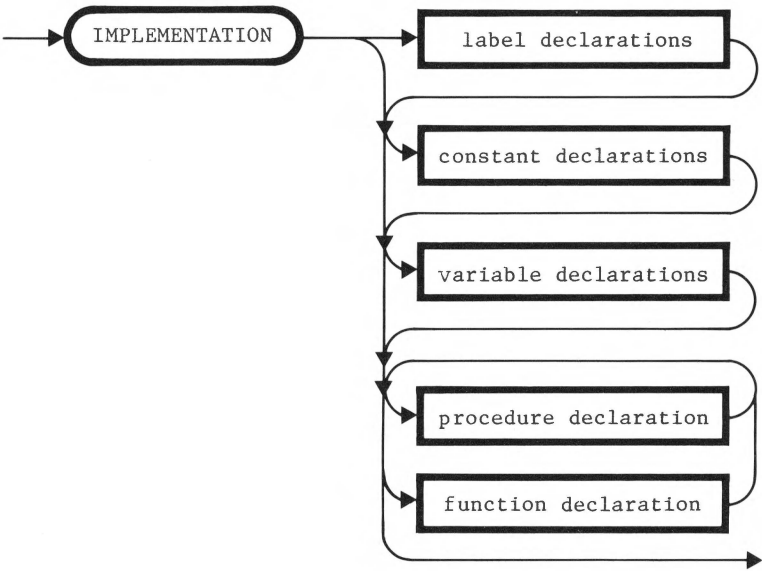
                  INTERFACE

VAR OFILRESULT, NFILRESULT: INTEGER;
{Two public variables:  these are both initialized to -1,
 and subsequently used by the public procedures to hold
 results from the IORESULT function.  Because of these
 variables, the data segment is needed.}
```

```
PROCEDURE OPENOLDFILE (VAR F:TEXT; VAR PATHNAME:STRING);
PROCEDURE OPENNEWFILE (VAR F:TEXT; VAR PATHNAME:STRING);
{Two public procedures for opening text files:  these are
 similar except that OPENOLDFILE uses RESET while
 OPENNEWFILE uses REWRITE.  If the pathname has no suffix,
 the suffix .TEXT is added.  I/O checking is turned off
 while the file is opened, and then IORESULT is used to
 check whether the file was opened successfully.
 OPENOLDFILE puts the result in OFILRESULT and OPENNEWFILE
 puts the result in NFILRESULT; the host program can check
 these variables to find out what happened.}
```

```
                     IMPLEMENTATION
```

```
{A private procedure that is called by the public ones, to
 add the .TEXT suffix to a pathname if it doesn't have a
 suffix:}
PROCEDURE ADDSUFF(VAR PATHNAME:STRING);
  BEGIN
    IF POS('.TEXT', PATHNAME) = Ø
      THEN PATHNAME := CONCAT(PATHNAME, '.TEXT')
  END;
```

```
{The first of the public procedures.  The parameter list is
 made into a comment, since the parameters have already been
 declared in the interface:}
PROCEDURE OPENOLDFILE {(VAR F:TEXT; VAR PATHNAME:STRING)};
  BEGIN
    ADDSUFF(PATHNAME);
    {$IOCHECK-}
    RESET(F, PATHNAME);
    {$IOCHECK+}
    OFILRESULT := IORESULT
  END;
```

```
{The other public procedure.  Again, the parameter list is
 made into a comment:}
PROCEDURE OPENNEWFILE {(VAR F:TEXT; VAR PATHNAME:STRING)};
  BEGIN
    ADDSUFF(PATHNAME);
    {$IOCHECK-}
    REWRITE(F, PATHNAME);
    {$IOCHECK+}
    NFILRESULT := IORESULT
  END;
```

{The initialization, which provides initial values for the two public variables. The value −1 is chosen because it is never returned by the IORESULT function:}
BEGIN
  NFILRESULT := −1;
  OFILRESULT := −1
END.

Using the Example Unit

The OPENS unit shown above can be compiled exactly as if it were a program. Then it can be installed in SYSTEM.LIBRARY , using the LIBRARY utility. Once in the library, the unit can be used by any Pascal host program. A sample program to use our unit is sketched out below:

```
PROGRAM USEIT;
USES OPENS;

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

BEGIN
  {At this point, OFILRESULT and NFILRESULT have both been
   initialized to −1.}
  WHILE OFILRESULT <> Ø DO BEGIN
    WRITE('Type input file name: ');
    READLN(INNAME);
    OPENOLDFILE(INFILE, INNAME);

    {At this point, INNAME has had the .TEXT suffix added
     if necessary, and OFILRESULT contains the result from
     IORESULT.}

    IF OFILRESULT <> Ø THEN
      WRITELN('Can''t open file ', INNAME)
  END;
```

```
    WHILE NFILRESULT <> Ø DO BEGIN
      WRITE('Type output file name: ');
      READLN(OUTNAME);
      OPENNEWFILE(OUTFILE, OUTNAME);

      {At this point, OUTNAME has had the .TEXT suffix added
       if necessary, and NFILRESULT contains the result from
       IORESULT.}

      IF NFILRESULT <> Ø THEN
        WRITELN('Can''t open file ', OUTNAME)
    END;
    ...
    CLOSE(INFILE);
    CLOSE(OUTFILE, LOCK)
  END.
```

The USES declaration must immediately follow the heading of the host program; procedures and functions may not contain their own USES declarations. At the occurrence of a USES declaration, the Compiler references the INTERFACE 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 to the host program.

A conflict will occur if the host program declares an identifier that is also publicly declared in 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. However, an intrinsic unit cannot use a regular unit. For example, a unit named MUSIC might use the APPLESTUFF unit (which is an intrinsic unit):

```
    UNIT MUSIC;
    INTRINSIC CODE 25;

                     INTERFACE

    USES APPLESTUFF;
    ...
```

When a host program uses a unit that uses another unit, the host program must declare that it uses the deepest nested unit first:

```
PROGRAM PLAYER;
    USES APPLESTUFF, MUSIC;
```

The following example is a unit that uses the OPENS unit shown previously.

```
UNIT OPENTXT;
INTRINSIC CODE 19 DATA 2∅;

                    INTERFACE

{The USES declaration must appear here:}
USES OPENS;


VAR FFF:TEXT;
{The variable FFF is a file used only by a private function;
 however it has to be declared here because private files
 are not allowed.  This is why this unit needs a data
 segment.}

PROCEDURE OPENINFILE(VAR INFILE:TEXT);
PROCEDURE OPENOUTFILE(VAR OUTFILE:TEXT);
{These are the two public procedures for opening textfiles
 for input and output.  They interact with the user to
 obtain pathnames, and use the procedures and variables in
 the OPENS unit.  They allow the user to exit from the
 program by pressing the RETURN key without typing a
 pathname.  OPENOUTFILE also checks to see if the pathname
 is that of an existing file, and asks the user to confirm
 that the existing file is to be destroyed.}
```

IMPLEMENTATION

{This private function is used to find out whether a
 pathname is the pathname of an existing file, by opening it
 with OPENOLDFILE. If the file exists, it will be
 successfully opened and the value of OFILRESULT will be ∅.
 The file is then closed.}
FUNCTION FILEXISTS(VAR NAME:STRING): BOOLEAN;
  BEGIN
    OPENOLDFILE(FFF, NAME);
    FILEXISTS := OFILRESULT = ∅;
    CLOSE(FFF)
  END;

{This is the public procedure to open a textfile for input.
 The parameter list is made into a comment.}
PROCEDURE OPENINFILE{(VAR INFILE:TEXT)};
  VAR INNAME: STRING;
  BEGIN
    REPEAT
      WRITE('Type input filename (RETURN to quit): ');
      READLN(INNAME);
      IF LENGTH(INNAME) = ∅
        THEN EXIT(PROGRAM)
        ELSE OPENOLDFILE(INFILE, INNAME);
      IF OFILRESULT <> ∅
        THEN WRITELN('Can''t open file ', INNAME)
    UNTIL OFILRESULT = ∅
  END;

{This is the public procedure to open a textfile for output.
The parameter list is made into a comment.}
PROCEDURE OPENOUTFILE{(VAR OUTFILE:TEXT)};
   VAR CH: CHAR;
       OUTNAME:STRING;
    BEGIN
      REPEAT
        WRITE('Type output filename (RETURN to quit): ');
        READLN(OUTNAME);
        IF LENGTH(OUTNAME) = Ø
          THEN EXIT(PROGRAM)
          ELSE BEGIN
            IF FILEXISTS(OUTNAME)
              THEN BEGIN
                WRITE('Destroy existing file ', OUTNAME, '? ');
                REPEAT
                  READ(KEYBOARD, CH)
                UNTIL CH IN ['Y', 'y', 'N', 'n'];
                WRITELN(CH);
                IF CH IN ['Y', 'y']
                  THEN OPENNEWFILE(OUTFILE, OUTNAME)
              END
              ELSE OPENNEWFILE(OUTFILE, OUTNAME)
          END
      UNTIL NFILRESULT = Ø
    END;

{This unit needs no initialization, so it ends with just the
word END followed by a period.}
END.

The following program uses the OPENINFILE and OPENOUTFILE
procedures in the OPENTXT unit:

PROGRAM FILEUSER;
USES OPENS, OPENTXT;

VAR INFILE, OUTFILE: TEXT;

BEGIN
  OPENINPUT(INFILE);
  OPENOUTPUT(OUTFILE);
  ...
  CLOSE(INFILE);
  CLOSE(OUTFILE, LOCK)
END.

Program Libraries and SYSTEM.LIBRARY

For any program, you can create a <u>program library</u> file to contain
units used by that program. When the program is executed, it can
then use intrinsic units from both the program library and
SYSTEM.LIBRARY .

By definition, the program library for any particular program is
a library file whose pathname is identical to the pathname of the
program's codefile, but with the suffix .LIB instead of .CODE.

For example, suppose that /USEFUL/SORTER.CODE is the codefile's
pathname. You can create a program library for this program by
using the pathname /USEFUL/SORTER.LIB (note that this means it is
on the same diskette).

In order to compile the program using units from the program
library, you must use the Compiler's USING option; see
Appendix F.

When any program is executed, each unit that it uses is searched
for first in the program library, if there is one, and then in
the SYSTEM.LIBRARY file on the system diskette.

If the local filename of the codefile is more than 11
characters and does not end in .CODE , then the system
forms the program library name by truncating to 11
characters and adding the .LIB suffix. For example, if
you have a codefile named /MINE/SYSTEM.STARTUP , the system
will look for a program library named
/MINE/SYSTEM.STAR.LIB .

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 run an unlinked code file with the R command will cause the Linker to run automatically, looking for the unit in the system library. Trying to execute an unlinked file with the X command 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.

15

Program Segmentation

The information in this chapter is not needed for small programs,
but can be crucial for large programs.

Program Segments

To make the most efficient use of the memory space available for
program code and data, programs can be divided into segments.
This section gives essential information on how the Pascal System
implements segmentation.

A segment is code (or data space) that can be loaded into memory
by itself, independent of other segments. Every program consists
of at least one segment, and some programs consist of many
segments. Whenever a program is compiled, the Compiler and
Linker create the following segments in the code file:

  - Each SEGMENT procedure or function becomes a segment in
    the code file.

  - Each regular unit that the program uses becomes a
    segment in the code file.

  - The main program itself becomes a segment in the code
    file. This includes the program's non-SEGMENT
    procedures and functions.

Similarly, whenever a regular unit is compiled, the result is a
code segment for the unit itself, plus an additional segment for
each regular unit that is used within the unit being compiled.
(Note that SEGMENT procedures and functions are not allowed
inside units.)

When an intrinsic unit is compiled, it produces a code segment,
and may produce a data segment as well. (Note that an intrinsic
unit cannot use a regular unit.)

Note that segments do not nest—every segment is just one segment
and does not contain any other segments. For example, if the
declaration of a SEGMENT procedure contains the declaration of
another SEGMENT procedure, the result is two distinct code
segments, even though they are nested syntactically and the scope
is nested.

The Segment Dictionary

Every code file (including library files) contains information called a segment dictionary. This contains an entry for each segment in the code file; the entry has all the information the system needs to load and execute the segment.

The segment dictionary has slots for 16 entries. Therefore, one code file can contain at most 16 segments. In the case of a program, this implies one segment for the program itself, one for each SEGMENT procedure or function, and one for each regular unit used by the program.

Note that intrinsic units used by a program do not require entries in the segment dictionary of the program's code file. This is because an intrinsic unit's code segment is never in the program's code file—it is in a library file, and appears in the library file's segment dictionary.

Therefore a program can have a maximum of 16 segments, not counting segments from intrinsic units. Counting segments from intrinsic units, a program can have up to 48 segments as explained below.

The Run-Time Segment Table

When a program is executed, the Pascal Interpreter uses a segment table which contains an entry for each segment that is used in executing the program. This table thus contains the following entries:

- Entries for 11 segments that the system uses when executing a user program

- An entry for each segment in the segment dictionary of the program's code file

- An entry for each intrinsic unit segment (both data and code segments).

The segment table has slots for up to 64 entries. Since the
system uses 11, this means that 53 slots are left for the program
to use. Remember that only 16 can be in the program's code file;
any excess over 16 must be intrinsic unit segments. Since each
intrinsic unit must be in either the program library or the
SYSTEM.LIBRARY file, which are code files, there can only be 32
intrinsic units altogether. Thus the maximum number of segments
a program can have is 48.

Segment Numbers

A segment number is an index into the segment table; thus at run
time, every segment has a segment number in the range $\emptyset$..63 and
no two segments in the program can have the same number.

These segment numbers are assigned to the program segments
(except intrinsic unit segments) when the segment entries are
placed in the code file's segment dictionary (before run time).
Numbers are assigned as follows:

  - The program itself is Segment 1.

  - The segments used by the system are $\emptyset$, 2..6, and
    59..63. These numbers are never assigned to segments
    of the program.

  - The segment numbers of regular unit segments and of
    SEGMENT procedures and functions are automatically
    assigned by the system; they begin at 7 and ascend.
    Note that after a regular unit is linked into a
    program, it may not have the same segment number that
    it had in the library.

The segment number of an intrinsic unit segment is determined by
the unit's heading, when the intrinsic unit is compiled. (These
numbers can be found by examining the segment dictionary of the
library file with the LIBMAP utility program.)

To summarize the above, the segment numbers of the program
itself, the segments used by the system, and any intrinsic units
used by the program are fixed before the program is compiled; the
segment numbers of regular units and of SEGMENT procedures and
functions are not fixed, and are assigned as the program is
compiled and linked, in ascending sequence beginning with 7.

Normally, the only time you need to specify segment numbers is in
writing an intrinsic unit, as explained in Chapter 14.

The NEXTSEG Option

When unavoidable segment-number conflicts arise there is a
solution: the Compiler has a NEXTSEG option which allows you to
specify the segment number of the next regular unit, SEGMENT
procedure, or SEGMENT function encountered by the Compiler.

Compiler options are commands that can be embedded in a Pascal
source file to control Compiler operation. Options are enclosed
in comment delimiters, and the first character within the comment
delimiters is the $ character. For complete details, see
Appendix F.

The NEXTSEG option has the form

```
    {$NS num}
 or {$NEXTSEG num}
```

where num is a literal integer constant that should be in the
range 8..56. The effect is to set the next segment number to
num.

The NEXTSEG option is ignored if it precedes the program heading;
this means that it cannot be used to specify the segment number
of the program itself.

The NEXTSEG option will only work if the specified number is
greater than the "default" number that would be automatically
assigned. If the number specified in the NEXTSEG option is less
than or equal to the default segment number, the option is
ignored.

For example, suppose that you want to use an intrinsic unit named
ZEBRA, whose code segment number is 7 and whose data segment
number is 8. (Normally, such numbers should be avoided in
writing intrinsic units.) Your program also contains a SEGMENT
procedure:

```
    PROGRAM ELEPHANT;
    USES ZEBRA;
    ...
    SEGMENT PROCEDURE HORSE;
    ...
```

The Compiler will automatically compile the HORSE procedure as
segment number 7, and when you try to execute the program, the
Pascal system will not execute the codefile because the program
has two different segments with the number 7.   There are two
remedies: recompile ZEBRA with different segment numbers (if you
have the source for ZEBRA) or use the NEXTSEG option in your
program:

            PROGRAM ELEPHANT;
            USES ZEBRA;
            ...
            {$NEXTSEG 9}
            SEGMENT PROCEDURE HORSE;
            ...

Now HORSE will become segment 9 instead of segment 7, and the
conflict is avoided.

Loading of Segment Procedures and Functions

Normally, the code of a SEGMENT procedure or function is present
in memory only as long as it is active.   If it is not active when
it is called, it is loaded from the code file (on diskette).   The
following program illustrates this:

            PROGRAM ONE; {Segment ONE is always in memory.}

                SEGMENT PROCEDURE ALPHA; {In memory only when active.}
                BEGIN
                ...
                END;

```
SEGMENT PROCEDURE BRAVO; {In memory only when active.}
  SEGMENT PROCEDURE CHARLIE; {In memory only when active.}
    BEGIN {Body of CHARLIE}
      ...
      ALPHA; {When this is executed, the segments in
              memory are ONE, ALPHA, BRAVO, and CHARLIE.}
      ...
    END;
  BEGIN {Body of BRAVO}
    ...
    CHARLIE; {When this starts executing, the segments in
              memory are ONE, BRAVO, and CHARLIE.}
    ALPHA; {When this is executed, the segments in
            memory are ONE, BRAVO, and ALPHA.}
    ...
  END;

BEGIN {Body of ONE}
  ...
  ALPHA; {When this is executed, the segments in
          memory are ONE and ALPHA.}
  BRAVO; {When this starts executing, the segments in
          memory are ONE and BRAVO.}
  ...
END.
```

The RESIDENT option can be used to alter this, as explained
below.

Loading of Unit Segments

Normally, all segments of units used by a program are loaded
automatically before the program begins executing, and remain in
memory throughout program execution. For example, consider the
following program where DELTA and GAMMA are two units, either
regular or intrinsic:

```
PROGRAM TWO
USES DELTA, GAMMA;
...
BEGIN
  ...
END.
```

Throughout program execution, the segments in memory are TWO, DELTA, and GAMMA. This can be altered by the NOLOAD option, as explained below. In any case, the initialization code for every intrinsic unit is executed at program startup time. The order in which unit names are listed in the USES statement at the beginning of the program is significant; the initialization code for the units is executed in this order.

The NOLOAD Option

The NOLOAD option has the form

```
    {$N+}
 or {$NOLOAD+}
```

The option is placed at the beginning of the main program body (after the BEGIN). It causes all unit segments to be swapped in and out in the same way as SEGMENT procedures: thus a unit segment is in memory only when a procedure or function in its INTERFACE is referenced by the program.

The {$NOLOAD+} option does not prevent the initialization of a unit from being loaded and executed before program execution; but after initialization the unit segment is unloaded until it is activated. Also, the initialization code is <u>not</u> executed when the unit is reloaded.

Consider the following program, where HUGEPROC is a large SEGMENT procedure and BIGUNIT is a large unit. The system does not have enough memory to hold HUGEPROC and BIGUNIT at the same time, along with the program itself.

```
PROGRAM THREE;
USES BIGUNIT;

SEGMENT PROCEDURE HUGEPROC;
  BEGIN
    •••
  END;

BEGIN
  {$NOLOAD+}  {Keeps BIGUNIT out of memory until needed.}
  HUGEPROC;
  •••
  CALCULATE;  {A procedure in BIGUNIT}
  •••
  HUGEPROC
END.
```

First HUGEPROC is called; BIGUNIT is not in memory because of the
{$NOLOAD+} option.   When CALCULATE is called, HUGEPROC is not in
memory since it is a SEGMENT procedure; it is immediately swapped
in.   As soon as no part of BIGUNIT is active, it is again swapped
out of memory, and HUGEPROC can be called again.

The RESIDENT Option

The RESIDENT option has one of the following forms:

```
{$R identifier}
{$RESIDENT identifier}
{$R number}
{$RESIDENT number}
```

where the identifier is the name of a unit or a SEGMENT procedure
or function, and the number is the segment number of a unit or a
SEGMENT procedure or function.   This unit, procedure, or function
is then said to be "resident" within the procedure or function
that contains the option.

The RESIDENT option is placed at the beginning of the body of a
procedure or function (after the BEGIN).   It alters the handling
of segments that would otherwise be in memory only when active:
that is, SEGMENT procedures and functions, and units under the
NOLOAD option.

When such a segment is called from a procedure or function that
specifies it to be resident, it is immediately loaded into memory
and remains there as long as the calling procedure or function is
active. For example, consider the following program:

```
PROGRAM FOUR;
USES BIGUNIT;

SEGMENT PROCEDURE HUGEPROC;
  BEGIN
    ...
  END;

PROCEDURE CALLHUGEPROC;
  VAR I: INTEGER;
  BEGIN
    FOR I:=1 TO 1ØØ DO HUGEPROC
  END;

PROCEDURE CALLCALCULATE;
  VAR I: INTEGER;
  BEGIN
    FOR I:=1 TO 1ØØ DO CALCULATE {A procedure in BIGUNIT}
  END;

BEGIN
  {$NOLOAD+} {Keeps BIGUNIT out of memory until needed.}
  HUGEPROC;
  ...
  CALCULATE;
  ...
  CALLHUGEPROC;
  ...
  CALLCALCULATE
END.
```

This resembles the previous example, but the CALLHUGEPROC and
CALLCALCULATE procedures are new. As written, these two
procedures have a problem: since HUGEPROC is a SEGMENT
procedure, it will be swapped in from diskette 1ØØ times when
CALLHUGEPROC executes, and because of the {$NOLOAD+} option in
the main program body, BIGUNIT will be swapped in 1ØØ times when
CALLCALCULATE executes. This is obviously undesirable, and it
can be prevented by using the RESIDENT option in each of these
procedures:

```
PROCEDURE CALLHUGEPROC;
  VAR I: INTEGER;
  BEGIN
  {$RESIDENT HUGEPROC}
    FOR I:=1 TO 1ØØ DO HUGEPROC
  END;
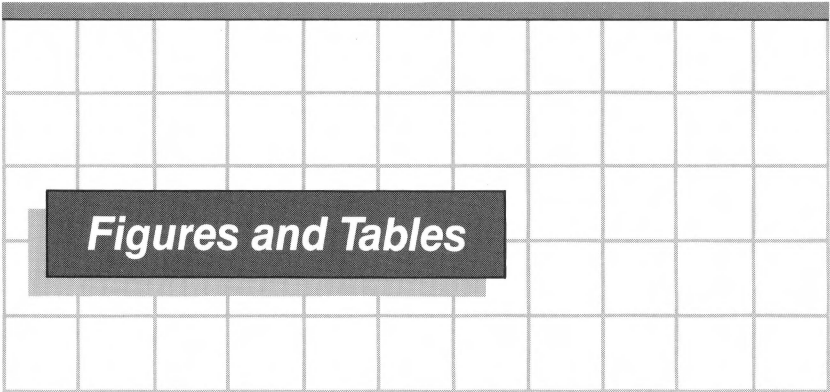
PROCEDURE CALLCALCULATE;
  VAR I: INTEGER;
  BEGIN
  {$RESIDENT BIGUNIT}
    FOR I:=1 TO 1ØØ DO CALCULATE {A procedure in BIGUNIT}
  END;
```

Now HUGEPROC will be kept in memory as long as CALLHUGEPROC is active, and BIGUNIT will be kept in memory as long as CALLCALCULATE is active.

Finally, note that the RESIDENT option can be applied to more than one segment, by separating instances of the options with commas as in the following example:

```
{$RESIDENT ALPHA,RESIDENT BETA,RESIDENT GAMMA}
{no spaces after commas}
```

where ALPHA, BETA, and GAMMA are names of segments (units, SEGMENT procedures, or SEGMENT functions). The option shown would make all three segments resident in the procedure containing the option.

Figures and Tables

Volume I—Chapters

1 What is Apple III Pascal?

2 Overview of Pascal

3 Simple Data Types

4 *Expressions and Assignments*

5 *The Flow of Control*

6 *Procedures and Functions*

7 Arrays, Sets, and Strings

8 Records

9 Pointers and Dynamic Variables

10 Introduction To Files and I/O

11 Text I/O

12 *Block File I/O and Device I/O*

13 *Special-Purpose Built-Ins*

14 *Library Units*

15 *Program Segmentation*

Volume II—Appendices

A The TRANSCEND and REALMODES Units

B The PGRAF Unit

C The CHAINSTUFF Unit

D The APPLESTUFF Unit

E Floating-Point Arithmetic

F   The Apple III Pascal Compiler

G   Special Techniques

H   Comparison To Apple II Pascal

I   Syntax Diagrams

J *Tables*

K *The TURTLEGRAPHICS Unit*

Index

The page numbers in this index do not refer to every occurrence of a word or phrase in the text. Instead, they refer to the locations of significant information on the topic related to the word or phrase.

References in Volume II are shown in square brackets [ ].

F

G

Tuck end flap
inside back cover
when using manual.

Apple III Pascal: *Programmer's Manual Volume 1*