Burroughs

Reference Manual

B 20 Systems

Pascal

(Relative to Release Level 3.0)

Priced Item
Printed in U.S.A.
August 1983

1162955

Burroughs

B 20 Systems Pascal

(Relative to Release Level 3.0)

Priced Item
Printed in U.S.A.
August 1983

1162955

Burroughs cannot accept any financial or other responsibilities that may be the result of your use of this information or software material, including direct, indirect, special or consequential damages. There are no warranties extended or granted by this document or software material.

You should be very careful to ensure that the use of this software material and/or information complies with the laws, rules, and regulations of the jurisdictions with respect to which it is used.

The information contained herein is subject to change without notice. Revisions may be issued to advise of such changes and/or additions.

Correspondence regarding this publication should be forwarded using the Remarks form at the back of the manual, or may be addressed directly to Documentation East , Burroughs Corporation, P.O. Box CB7, Malvern, Pennsylvania, 19355, U.S. America.

LIST OF EFFECTIVE PAGES

Page	Issue
iii	Original
iv	Blank
v thru xi	Original
xii	Blank
1-1 thru 1-4	Original
2-1 thru 2-10	Original
3-1 thru 3-5	Original
3-6	Blank
4-1 thru 4-15	Original
4-16	Blank
5-1 thru 5-14	Original
6-1 thru 6-38	Original
7-1 thru 7-12	Original
8-1 thru 8-13	Original
8-14	Blank
9-1 thru 9-16	Original
10-1 thru 10-18	Original
11-1 thru 11-27	Original
11-28	Blank
12-1 thru 12-22	Original
13-1 thru 13-14	Original
14-1 thru 14-12	Original
A-1 thru A-3	Original
A-4	Blank
B-1 thru B-6	Original
C-1 thru C-11	Original
C-12	Blank
D-1 thru D-46	Original
E-1 thru E-3	Original
E-4	Blank
F-1 thru F-4	Original
G-1 thru G-3	Original
G-4	Blank
H-1 thru H-9	Original
H-10	Blank
I-1 thru I-7	Original
I-8	Blank

TABLE OF CONTENTS

TABLE OF CONTENTS (CONT.)

TABLE OF CONTENTS (CONT.)

CHAPTER 1
INTRODUCTION AND FEATURES

CONTENTS

OVERVIEW

This document provides a complete description of a highly extended version of Pascal, as implemented for use on the BURROUGHS B 20. It is assumed that you have a working familiarity with Pascal. The Pascal described in this manual is highly portable and is consistent with the International Standard Organization (ISO) standard.

Unlike many other compilers which produce intermediate p-code for microcomputers, this Pascal Compiler generates native machine code. Programs compiled to native code execute much faster than those compiled to p-code. Thus, with this Pascal Compiler, you get the programming advantages of a high-level language without sacrificing execution speed. Because of many low-level escapes to the machine level, programs written in this Pascal are often comparable in speed to programs written in assembly language.

PASCAL LEVELS

This Pascal is organized into three "levels", Standard, Extended, and System.

Standard Level

All Standard ISO Pascal programs are intended to compile and run correctly using this compiler. All of the extensions to the language are provided in Appendix H, of this manual.

Extended Level

Pascal intended for use on your system enhances ISO Pascal and is intended for structured and relatively safe extensions such as OTHERWISE in the CASE statement and the construction of the BREAK statement.

System Level

This level includes all features at the extended level as well as unstructured, machine oriented extensions that are either useful or necessary for system programming tasks. These additional extensions include the address types and access to all File Control Block fields.

In addition to the above language levels, the Pascal compiler recognizes requests to specify the kind of error checking to be generated. These are included in the Pascal "Metacommands".

PASCAL FEATURES

The following list includes just some of the features available

at the extended and system levels of this Pascal. These features are described in more detail later in this manual.

1. Underscore in identifiers, which improves readability.

2. Nondecimal numbering (hexadecimal, octal, and binary), which facilitates programming at the byte and bit level.

3. Structured constants, which may be declared in the declaration section of a program or use in statements.

4. Variable length strings (type LSTRING), as well as special predeclared procedures and functions for LSTRINGs, which overcome standard Pascal's string handling capabilities.

5. Super arrays, a special variable length array whose declaration permits passing arrays of different lengths to a reference parameter, as well as dynamic allocation of arrays of different lengths.

6. Predeclared unsigned BYTE (Ø-255) and WORD (Ø-65535) types, which facilitate programming at the system level.

7. Address types (segmented and unsegmented), which allow manipulation of actual machine addresses at the system level.

8. String reads, which allow the standard procedures READ and READLN to read strings as structures rather than character by character.

9. Interface to assembly language, provided by PUBLIC and EXTERN procedures, functions, and variables, which allows low-level interfacing to assembly language and library routines.

1Ø. VALUE section, where you may declare the initial constant values of variables in a program.

11. Function return values of a structured type as well as of a simple type.

12. Direct (random access) files, accessible with the SEEK procedure, which enhance standard Pascal's file accessing capabilities.

13. Lazy evaluation, a special internal mechanism for interactive files that allows normal interactive input from terminals.

14. Structured BREAK and CYCLE statements, which allow structured exits from a FOR, REPEAT, or WHILE loop; and the RETURN statement, which allows a structured exit from a procedure or function.

15. OTHERWISE in CASE statements, where by you avoid explicitly

specifying each CASE constant.

16. STATIC attribute for variables, which allows you to indicate that a variable is to be allocated at a fixed location in memory rather than on the stack.

17. ORIGIN attribute, which may be given to variables, procedures, and functions to indicate their absolute location in memory.

19. Separate compilation of portions of a program (units and modules).

20. Conditional compilation, using conditional metacommands in your Pascal source file to switch on or off compilation of parts of the source.

CHAPTER 2
LANGUAGE OVERVIEW

CONTENTS

The Pascal language includes a large number of inter-related components. To ease your understanding of the language, its basic elements are discussed first. Each component is discussed in relation to their next higher-level component.

PASCAL NOTATION

All Pascal programs consist of an irreducible set of symbols with which the higher syntactic components of the language are created. The underlying notation is the ASCII character set, divided into the following syntactic groups:

1. Identifiers are the names given to individual instances of components of the language.

2. Separators are characters that delimit adjacent numbers, reserved words, and identifiers.

3. Special symbols include punctuation, operators, and reserved words.

4. Some characters are unused but are available for use in a comment or string literal.

METACOMMANDS

The metacommands provide a control language for the Pascal Compiler. The metacommands let you specify options that affect the overall operation of a compilation. For example, you can conditionally compile different source files, generate a listing file, or enable or disable runtime error checking code.

Metacommands are inserted inside comment statements. All of the metacommands begin with a dollar sign ($). Some may also be given as switches when the compiler is invoked.

Although most implementations of Pascal have some type of compiler control, the metacommands listed below are not part of standard Pascal and hence are not portable.

The metacommands are listed below:

$BRAVE
$DEBUG
$ENTRY
$ERRORS
$GOTO
$INCLUDE
$INCONST
$PAGE
$PAGEIF
$PAGESIZE
$POP
$PUSH
$RANGECK
$REAL

```
$INDEXCK                    $ROM
$INITCK                     $RUNTIME
$IF $THEN $ELSE $END        $SIMPLE
$INTEGER                    $SIZE
$LINE                       $SKIP
$LINESIZE                   $SPEED
$LIST                       $STACKCK
$MATHCK                     $SUBTITLE
$MESSAGE                    $SYMTAB
$NILCK                      $TITLE
$OCODE                      $WARN
$OPTBUG
```

See Chapter 4, "Metacommands," for a complete discussion of metacommands.

IDENTIFIERS AND CONSTANTS

Identifiers are names that denote the constants, variables, data types, procedures, functions, and other elements of a Pascal program.

An identifier must begin with a letter (A through Z or a through z). The initial letter may be followed by any number of letters, digits (Ø through 9), or underscore characters.

The compiler ignores the caseof letters; thus, "A" and "a" are equivalent. The only restriction on identifiers is that you must not choose a Pascal reserved word (see Chapter 3, Pascal Notation for a discussion of reserved words or Appendix E," Summary of Reserved Words," for a complete list).

A constant is a value that is not expected to change during the course of a program. A constant may be:

1. a number, such as 1.234 and 1ØØ
2. a string enclosed in single quotation marks, such as 'Miracle' or 'A12Ø7'
3. a constant identifier that is a synonym for a numeric or string constant

You can declare constant identifiers in the CONST section of a compiland, procedure, or function

```
CONST  REAL_CONST = 1.234;
       MAX_VAL    = 1ØØ;
       TITLE      = 'Pascal';
```

You can declare constants anywhere in the declaration section of a compilable part of a program, any number of times. Two

powerful extensions in Pascal are structured constants and constant expressions.

1. VECTOR, in the following example, is an array constant:

```
CONST VECTOR = VECTORTYPE (1,2,3,4,5);
```

2. MAXVAL, in the following example, is a constant expression (A, B, C, and D must also be constants):

```
CONST MAXVAL = A * (B DIV C) + D - 5;
```

DATA TYPES

Much of Pascal's power and flexibility lies in its data typing capability. The data types can be divided into three broad categories: simple, structured, and reference types.

1. A simple data type represents a single value; a structured type represents a collection of values. The simple types include the following:

INTEGER	enumerated
WORD	subrange
CHAR	REAL
BOOLEAN	INTEGER4

2. The structured data types include the following:

```
ARRAY
RECORD
SET
FILE
```

3. Reference types allow recursive definition of types in an extremely powerful manner.

All variables in Pascal must be assigned a data type. A type is either predeclared (e.g., INTEGER and REAL) or defined in the declaration section of a program. The following sample type declaration creates a type that can store information about a student:

```
TYPE

   STUDENT = RECORD
      AGE        : 5..18;
      SEX        : (MALE, FEMALE);
      GRADE      : INTEGER;
      GRADE_PT   : REAL;
      SCHEDULE   : ARRAY [1..10] OF CLASSES
   END;
```

VARIABLES AND VALUES

A variable is a value that is expected to change during the course of a program. Every variable must be of a specific data type.

After you declare a variable in the heading or declaration section of a compiland, procedure, or function, it may be used in any of the following ways:

1. You may initialize it, in the VALUE section of a program.

2. You may assign it a value, with an assignment statement.

3. You may pass it as a parameter to a procedure or function.

4. You may use it in an expression.

The VALUE section is a feature that applies only to statically allocated variables (variables with a fixed address in memory). You must first declare the variables, as shown in the following example:

```
VAR I, J, K, L :  INTEGER;
```

and then assign them initial values in the VALUE section:

```
VALUE I := 1;  J := 2;  K := 3;  L := 4;
```

Later, in statements, the variables can be assigned to,and used as operands in expressions:

```
I := J + K + L;
J := 1 + 2 + 3;
K := (J * K) + 9 + (L DIV J);
```

EXPRESSIONS

An expression is a formula for computing a value. It consists of a sequence of operators (which indicate the action to be performed) and operands (the value on which the operation is performed). Operands may contain function invocations, variables, constants, or even other expressions. In the following expression, plus (+) is an operator, while A and B are operands:

```
A + B
```

There are three basic kinds of expressions:

1. Arithmetic expressions perform arithmetic operations on the operands in the expression.

2. Boolean expressions perform logical and comparison operations with Boolean results.

3. Set expressions perform combining and comparison operations on sets, with Boolean or set results.

Expressions always return values of a specific type. For instance, if A, B, C, and D are all REAL variables, then the following expression evaluates to a REAL result:

```
A * B + (C / D) + 12.3
```

Expressions may also include function designators:

```
ADDREAL (2, 3) + (C / D)
```

ADDREAL is a function that has been previously declared in a program. It has two REAL value parameters, which it adds together to obtain a total. This total is the return value of the function, which is then added to (C / D).

Expressions are not statements, but may be components of statements. In the following example, the entire line is a statement; only the portion after the equal sign is an expression:

```
X := 2 / 3 + A * B
```

STATEMENTS

Statements perform actions, such as computing, assigning, altering the flow of control, and reading and writing files. Statements are found in the bodies of programs, procedures, and functions and are executed as a program runs.

Statements in Pascal are as follows:

Statement	**Purpose**
Assignment	Replaces the current value of a variable with a new value.
BREAK	Exits the currently executing loop.
CASE	Allows for the selection of one action from a choice of many, based on the value of an expression.
CYCLE	Starts the next iteration of a loop.

FOR	Executes a statement repeatedly while a progression of values is assigned to a control variable.
GOTO	Continues processing at another part of the program.
IF	Together with THEN and ELSE, allows for conditional execution of a statement.
Procedure call	Invokes a procedure with actual parameter values
REPEAT	Repeats a sequence of statements one or more times, until a Boolean expression becomes true.
RETURN	Exits the current procedure, function, program, or implementation.
WHILE	Repeats a statement zero or more times, until a Boolean expression becomes false.
WITH	Opens the scope of a statement to include the fields of one or more records, so that you can refer to the fields directly.

PROCEDURES AND FUNCTIONS

Procedures and functions act as subprograms that execute under the supervision of a main program. However, unlike programs, procedures and functions can be nested within each other and can even call themselves. Furthermore, they have sophisticated parameter-passing capabilities that programs lack.

Procedures are invoked as statements; functions can be invoked in expressions wherever values are called for. A procedure declaration, like a program, has a heading, a declaration section, and a body.

Example of a procedure declaration:

```
PROCEDURE COUNT_TO(NUM : INTEGER);  {Heading}

VAR I : INTEGER;                    {Declaration section}

BEGIN                               {Body}
  FOR I := 1 TO NUM DO
  WRITELN (I)
END;
```

A function is a procedure that returns a value of a particular type; hence, a function declaration must indicate the type of the return value.

Example of a function declaration:

```
FUNCTION ADD (VAL1, VAL2 : INTEGER): INTEGER;   {Heading}
BEGIN                                           {Body}
   ADD := VAL1 + VAL2
END;
```

Procedures and functions look somewhat different from programs, in that their parameters have types and other options. Like the body of a program, the body of a procedure or a function is enclosed by the reserved words BEGIN and END; however, a semicolon rather than a period follows the word "END".

Declaring a procedure or function is entirely distinct from using it in a program. The procedure and function declared above might actually appear in a program as follows:

```
TARGET_NUMBER := ADD (5, 6);   {Function ADD}
COUNT_TO (TARGET_NUMBER);      {Procedure COUNT_TO}
```

COMPILANDS

The Pascal Compiler processes programs, modules, and implementations of units. Collectively, these compilable programs and parts of programs are referred to as compilands. You can compile modules and implementations of units separately and then later link them to a program without having to recompile the module or unit.

The fundamental unit of compilation is a program. A program has three parts:

1. The program heading identifies the program and gives a list of program parameters.

2. The declaration section follows the program heading and contains declarations of labels, constants, types, variables, functions, and procedures. Some of these declarations are optional.

3. The body follows all declarations. It is enclosed by the reserved words BEGIN and END and is terminated by a period. The period is the signal to the compiler that it has reached the end of the source file.

The following program illustrates this three-part structure:

```
PROGRAM FRIDAY (INPUT,OUTPUT); {Program header}
```

```
LABEL 1;                              {Declaration section}
CONST DAYS_IN_WEEK = 7;
TYPE  KEYBOARD_INPUT = CHAR;
VAR   KEYIN : KEYBOARD_INPUT;

BEGIN                                 {Program body}
   WRITE('IS TODAY FRIDAY? ');
1: READLN(KEYIN);
   CASE KEYIN OF
    'Y', 'y' : WRITELN('It''s Friday.');
    'N', 'n' : WRITELN('It''s not Friday.');
   OTHERWISE
      WRITELN('Enter Y or N.');
      WRITE('Please re-enter: ');
      GOTO 1
   END
END.
```

This three-part structure (heading, declaration section, body) is used throughout the Pascal language. Procedures, functions, modules, and units are all similar in structure to a program.

Modules are program-like units of compilation that contain the declaration of variables, constants, types, procedures, and functions, but no program statements. You can compile a module separately and later link it to a program, but it cannot be executed by itself.

Example of a module:

```
MODULE MODPART;    {Module heading}


CONST PI = 3.14;   {Declaration section}

PROCEDURE PARTA;
   BEGIN
     WRITELN ('parta')
   END;


END.
```

A module, like a program, ends with a period. Unlike a program, a module contains no program statements.

A unit has two sections: an interface and an implementation. Like a module, an implementation may be compiled separately and later linked to the rest of the program. The interface contains the information that lets you connect a unit to other units, modules, and programs.

Example of a unit:

```
INTERFACE;                    {Heading for interface}
UNIT MUSIC (SING, TOP);

VAR TOP : INTEGER;            {Declarations for interface}
PROCEDURE SING;
BEGIN                         {Body of interface}
END;

IMPLEMENTATION OF MUSIC;      {Heading for implementation}

PROCEDURE SING;               {Declaration for implementation}

VAR I : INTEGER;

BEGIN
   FOR I := 1 TO TOP DO
   BEGIN
     WRITE ('FA '); WRITELN ('LA LA')
   END
END;


BEGIN                         {Body of implementation}
   TOP := 5
END.
```

A unit, like a program or a module, ends with a period. Modules and units let you develop large structured programs that can be broken into parts. This can be advantageous in the following situations:

1. If a program is large, breaking it into parts makes it easier to develop, test, and maintain.

2. If a program is large and recompiling the entire source file is time consuming, breaking the program into parts saves compilation time.

3. If you intend to include certain routines in a number of different programs, you can create a single object file that contains these routines and then link it to each of the programs in which the routines are used.

4. If certain routines have different implementations, you might place them in a module to test the validity of an algorithm and later create and implement similar routines in assembly language to increase the speed of the algorithm.

CHAPTER 3
PASCAL NOTATION

CONTENTS

All components of the Pascal language are constructed from the standard ASCII character set. Characters make up lines, each of which is separated by a character specific to the operating system. Lines make up files. Within a line, individual characters or groups of characters fall into one (or more) of four broad categories:

1. components of identifiers
2. separators
3. special symbols
4. unused characters

COMPONENTS OF IDENTIFIERS

Identifiers are names that denote the constants, variables, data types, procedures, functions, and other elements of a Pascal program. Identifiers must begin with a letter; subsequent components may include letters, digits, and underscore characters. Identifiers can be of any length, but must fit on a line. Only the first 31 characters are significant.

Letters

In identifiers, only the uppercase letters A through Z are significant. You may use lowercase letters for identifiers in a source program. However, the Pascal Compiler converts all lowercase letters in identifiers to the corresponding uppercase letters.

Letters in comments or in string literals may be either uppercase or lowercase; no mapping of lowercase to uppercase occurs in either comments or string literals.

Digits

Digits in Pascal are the numbers zero through nine. Digits can occur in identifiers such as AS129M, or in numeric constants such as 1.23 and 456.

The Underscore Character

The underscore (_) is the only nonalphanumeric character allowed in identifiers. You can use it like a space to improve readibility.

SEPARATORS

Separators delimit adjacent numbers, reserved words, and identifiers. A separator can be any of the following:

1. the space character
2. the tab character
3. the form feed character
4. the new line marker
5. the comment

Comments in take one of these forms:

```
{This is a comment, enclosed in braces.}
(*This is an alternate form of comment.*)
```

You can also have comments that begin with an exclamation point:

```
! The rest of this line is a comment.
```

For comments in this form, the new line character delimits the comment. Nested comments are permitted, so long as each level has different delimiters. Thus, when a comment is started, the compiler ignores succeeding text until it finds the matching end-of-comment.

SPECIAL SYMBOLS

Special symbols can be divided into:

1. punctuation
2. operators
3. reserved words

Punctuation

Punctuation serves a variety of purposes, including the following:

Symbol	Purpose
{ }	Braces delimit comments.
[ ]	Brackets delimit array indices, sets, and attributes. They may also replace the reserved words BEGIN and END

in a program.

( ) Parentheses delimit expressions, parameter lists, and program parameters.

' Single quotation marks enclose string literals.

:= The colon-equals symbol assigns values to variables in assignment statements and in VALUE sections.

; The semicolon separates statements and declarations.

: The colon separates variables from types and labels from statements.

= The equal sign separates identifiers and type clauses in a TYPE section.

, The comma separates the components of lists.

.. The double period denotes a subrange.

. The period designates the end of a program, indicates the fractional part of a real number, and also delimits fields in a record.

^ The up arrow denotes the value pointed to by a reference value.

The number sign denotes nondecimal numbers.

$ The dollar sign prefixes metacommands.

Operators

Operators are a form of punctuation that indicate some operation to be performed. Some are alphabetic, others are one or two nonalphanumeric characters. Operators that consist of more than one character must not have a separator between characters. The operators that consist only of nonalphabetic characters are the following:

```
+   -   *   /   >   <   =   <>   <=   >=
```

Some operators (e.g., NOT and DIV) are reserved words instead of nonalphabetic characters. (See Chapter 8, "Expressions," for a complete list of of the nonalphabetic operators and a discussion of the use of operators in expressions).

Reserved Words

Reserved words are a fixed part of the Pascal language. They include, for example, statement names (e.g., BREAK) and words like BEGIN and END that bracket the main body of a program. See Appendix E, "Summary of Pascal Reserved Words," for a complete list.

You cannot create an identifier that is the same as any reserved word. You may, however, declare an identifier that contains within it the letters of a reserved word (for example, the identifier DOT containing the reserved word DO).

UNUSED CHARACTERS

A few printing characters are not used in Pascal:

```
%   &   "   |   ~   `
```

You may, however, use them within comments or string literals. A number of other nonprinting ASCII characters will generate error messages if you use them in a source file other than in a comment or string literal:

1. the characters from CHR (Ø) to CHR (31), except the tab and form feed, CHR (9) and CHR (12), respectively

2. the characters from CHR (127) to CHR (255)

The tab character, CHR (9), is treated like a space and is passed on to the listing file. A form feed, CHR (12), is treated like a space and starts a new page in the listing file.

The ISO standard for ASCII reserves some character positions for national usage to permit larger alphabets, diacritical marks, and so on. Note that the number sign "#" is equivalent to the "pound" sign ("L" with a bar through it), as ASCII #23; also the currency symbol "$" is equivalent to the "scarab" sign (a circle with four spikes), as ASCII #24. The other 1Ø national symbols either are unused (#5C, #6Ø, #7C, #7E) or have substitutes available (@ #4Ø, [#5B,] #5D, ^ #5E, { #7B, } #7D).

CHAPTER 4
METACOMMANDS

CONTENTS

Metacommands make up the compiler control language. Metacommands are compiler directives that allow you to control such things as the following:

1. language level
2. debugging and error handling
3. optimization level
4. use of the source file during compilation
5. listing file format

You can specify one or more metacommands at the start of a comment; you should separate multiple metacommands with either spaces or commas. Spaces, tabs, and line markers between the elements of a metacommand are ignored. Thus, the following are equivalent:

```
{$PAGE:12}
{$PAGE : 12}
```

To disable metacommands within comments, you place any character that is not a tab or space in front of the first dollar sign, as shown:

```
{x$PAGE:12}
```

You may change compiler directives during the course of a program. For example, most of a program might use $LIST-, with a few sections using $LIST+ as needed. Some metacommands, such as $LINESIZE, normally apply to an entire compilation.

If you are writing Pascal programs for use with other compilers, keep in mind the fact that metacommands are always nonstandard and rarely transportable.

Metacommands invoke or set the value of a metavariable. Metavariables are classified as typeless, integer, on/off switch, or string.

1. Typeless metavariables are invoked when used, as in $PUSH.
2. Integer metavariables can be set to a numeric value, as in $PAGE:101.

3. On/off switches can be set to a numeric value so that a value greater than zero turns the switch on and a value equal or less than zero turns it off, as in $MATHCK:1.

4. String metavariables can be set to a character string value, such as with $TITLE:'COM PROGRAM'.

The following notations are used in metacommand descriptions in this chapter:

Notation	Meaning
	Metacommand is typeless.
+ or -	Metacommand is an on/off switch. + sets value to 1 (on). - sets value to Ø (off). Default is indicated by + or - in heading.
:<n>	Metacommand is an integer.
:'<text>'	Metacommand is a string.

String values in the metalanguage may be either a literal string or string constant identifier. Constant expressions are not allowed for either numbers or strings, although you can achieve the same effect by declaring a constant identifier equal to the expression and using the identifier in the metacommand.

In metacommands only, Boolean and enumerated constants are changed to their ORD values. Thus, a Boolean false value becomes Ø and true becomes 1.

For a complete alphabetical listing of Pascal metacommands see Appendix G, "Summary of Pascal Metacommands."

LANGUAGE LEVEL SETTING AND OPTIMIZATION

The following metacommands allow you to control the level (standard, extended, or system) at which the compiler processes your program and the degree to which optimization is used:

Name	Description
$ROM	Gives a warning on static initialization.
$SIMPLE	Disables global optimization.
$SIZE	Minimizes size of code generated.

$SPEED Minimizes execution time of code.

The metacommands $INTEGER and $REAL set the length (i.e., precision) of the standard INTEGER and REAL data types. $INTEGER can only be set to 2 (the default), for 16-bit integers. However, you may set $REAL to either 4, or 8 (the default), to make type REAL identical to REAL4 or REAL8, respectively.

The $SIMPLE turns off common subexpression optimization while $SIZE and $SPEED turn it back on. If $ROM is set, the compiler gives a warning that static data will not be initialized in either of the following situations:

1. at a VALUE section
2. every place where static data initialization occurs due to $INITCK (described under "Debugging and Error Handling")

DEBUGGING AND ERROR HANDLING

The following metacommands are for debugging and error handling. They also generate code to check for runtime errors:

Metacommand	**Description**
$BRAVE+	Sends error messages and warnings to the terminal screen.
$DEBUG-	Turns on or off all the debug checking (CK in metacommands below).
$ENTRY-	Generates procedure entry/exit calls for debugger.
$ERRORS:<n>	Sets number of errors allowed per page (default is 25).
$GOTO-	Flags GOTO statements as "considered harmful."
$INDEXCK+	Checks for array index values in range, including super array indices.
$INITCK-	Checks for use of uninitialized values.
$LINE-	Generates line number calls for the debugger.

$MATHCK+	Checks for mathematical errors such as overflow and division by zero.
NILCK+	Checks for bad pointer values.
RANGECK+	Checks for subrange validity.
RUNTIME-	Determines context of runtime errors.
STACKCK+	Checks for stack overflow at procedure or function entry.
WARN+	Gives warning messages in listing file.

If any check is on when the compiler processes a statement, tests relevant to the statement are done. A runtime error invokes a call to the runtime support routine, EMSEQQ (synonymous with ABORT). When EMSEQQ is called, the compiler passes the following information to it:

1. an error message
2. a standard error code
3. an operating system return code error status value

EMSEQQ also has available:

1. the program counter at the location of the error
2. the stack pointer at the location of the error
3. the frame pointer at the location of the error
4. the current line number (if $LINE is on)
5. the current procedure or function name and the source filename in which the procedure or function was compiled (if $ENTRY is on)

$BRAVE+

Sends error messages and warnings to your terminal (in addition to writing them to the listing file). If the number of errors and warnings is more than will fit on the screen, the earlier ones scroll off and you will have to check the listing file to see them all.

$DEBUG-

Turns on or off all of the debug switches (i.e., those that end with "CK"). You may find it useful to use $DEBUG- at the beginning of a program to turn all checking off and then selectively turn on only the debug switches you want. Alternatively, you may use this metacommand to turn all debugging on at the start and then selectively turn off those you don't need as the program progresses. By default, some error checks are on and some off.

$ENTRY-

Generates procedure and function entry and exit calls. This lets a debugger or error handler determine the procedure or function in which an error has occurred. Since this switch generates a substantial amount of extra code for each procedure and function, you should use it only when debugging. Note that $LINE+ requires $ENTRY+; thus, $LINE+ turns on $ENTRY, and $ENTRY- turns off $LINE.

$ERRORS:<n>

Sets an upper limit for the number of errors allowed per page. Compilation aborts if that number is exceeded. The default is 25 errors and/or warnings per page.

$GOTO-

Flags GOTO statements with a warning that they are "considered harmful." This warning may be useful in either of the following circumstances:

1. encourage structured programming in an educational environment
2. to flag all GOTO statements during the process of debugging

$INDEXCK

Checks that array index values, including super array indices, are in range. Since array indexing occurs so often, bounds checking is enabled separately from other subrange checking.

$INITCK-

Checks for the occurrence of uninitialized values, such as the following:

1. uninitialized INTEGERs and 2-byte INTEGER subranges with the hexadecimal value 16#8000

2. uninitialized 1-byte INTEGER subranges with the hexadecimal value 16#80

3. uninitialized pointers with the value 1 (if $NILCK is also on)

4. uninitialized REALs with a special value

The $INITCK metacommand generates code to perform the following actions:

1. set such values uninitialized when they are allocated

2. set the value of INTEGER range FOR-loop control variables uninitialized when the loop terminates normally

3. set the value of a function that returns one of these types uninitialized when the function is entered

$INITCK never generates any initialization or checking for WORD or address types. Statically allocated variables are loaded with their initial values. Also, $INITCK does not check values in an array or record when the array or record itself is used.

Variables allocated on the stack or in the heap are assigned initial values with generated code. $INITCK does not initialize any of the following classes of variables:

1. variables mentioned in a VALUE section

2. variant fields in a record

3. components of a super array allocated with the NEW procedure

$LINE-

Generates a call to a debugger or error handler for each source line of executable code. This allows the debugger to determine the number of the line in which an error has

occurred. Because this metacommand generates a substantial amount of extra code for each line in a program, you should turn it on only when debugging. Note that $LINE+ requires $ENTRY+, so $LINE+ turns on $ENTRY, and $ENTRY- turns off $LINE.

$MATHCK+

Checks for mathematical errors, including INTEGER and WORD overflow and division by zero. $MATHCK does not check for an INTEGER result of exactly -MAXINT-1 (i.e., #8000); $INITCK does catch this value if it is assigned and later used.

Turning $MATHCK off does not always disable overflow checking. There are, however, library routines that provide addition and multiplication functions that permit overflow (LADDOK, LMULOK, SADDOK, SMULOK, UADDOK, and UMULOK). For a description of each of these functions see chapter 11, Available Procedures and Functions.

$NILCK+

Checks for the following conditions:

1. dereferenced pointers whose values are NIL
2. uninitialized pointers if $INITCK is also on
3. pointers that are out of range
4. pointers that point to a free block in the heap

$NILCK occurs whenever a pointer is dereferenced or passed to the DISPOSE procedure. $NILCK does not check operations on address types.

$RANGECK+

Checks subrange validity in the following circumstances:

1. assignment to subrange variables
2. CASE statements without an OTHERWISE clause
3. actual parameters for the CHR, SUCC, and PRED functions
4. indices in PACK and UNPACK procedures

5. set and LSTRING assignments and value parameters

6. super array upper bounds passed to the NEW procedure

$RUNTIME-

If the $RUNTIME switch is on when a procedure or function is compiled, the "location of an error" is the place where the procedure or function was called rather than the location in the procedure or function itself. This information is normally sent to your terminal, but you could link in a custom version of EMSEQQ, the error message routine, to do something different (such as invoke the runtime debugger or reset a controller). For more information on error handling, see Appendix C, "Runtime Structure".

$STACKCK+

Checks for stack overflow when entering a procedure or function and when pushing parameters larger than four bytes on the stack.

$WARN+

Sends warning messages to the listing file (this is the default). If this switch is turned off, only fatal errors are printed in the source listing.

SOURCE FILE CONTROL

The following metacommands provide some measure of control over the use of the source file during compilation. These commands are listed in Table 17.4 and described in more detail below.

Name	Description
$IF constant $THEN <text1> $ELSE <text2> $END	Allows conditional compilation of <text1> source if <constant> is greater than zero.
$INCLUDE:'<filename>'	Switches compilation from current source file to source file named.
$INCONST:<text>	Allows interactive setting of constant values at

	compile time.
$MESSAGE:'<text>'	Allows the display of a message the terminal screen to indicate which version of a program is compiling.
$POP	Restores saved value of all metacommands.
$PUSH	Saves current value of all metacommands.

Because the compiler keeps one look-ahead symbol, it actually processes metacommands that follow a symbol before it processes the symbol itself. This characteristic of the compiler can be a factor in cases such as the following:

```
CONST Q = 1;
{$IF Q $THEN}      {Q is undefined in the $IF.}


CONST Q = 1; DUMMY = 0;
{$IF Q $THEN}      {Now Q is defined.}

X := P^;
{$NILCK+}          {NILCK applies to P^ here.}


X := P^;

{$NILCK-}          {NILCK doesn't apply to P.}
```

$IF <constant> $THEN <text> $END

Allows for conditional compilation of a source text. If the value of the constant is greater than zero, then source text following the $IF is processed; otherwise it is not. An $IF $THEN $ELSE construction is also available, as in the following example:

```
{$IF BTOS $THEN}
SECTOR = S12;
{$ELSE}
SECTOR  = S128;
{$END}
```

To simulate an $IFNOT construction, use the following form of the metacommand:

$IF <constant> $ELSE <text> $END

The constant may be a literal number or constant identifier. The text between $THEN, $ELSE, and $END is arbitrary; it can include line breaks, comments, other metacommands (including nested $IFs), etc. Any metacommands within skipped text are ignored, except, of course, corresponding $ELSE or $END metacommands.

Examples using the metaconditional:

```
{$IF  FPCHIP $THEN}
  CODEGEN (FADDCALL,T1,LEFTP)
{$END}
{$IF COMPSYS $ELSE}
  IF USERSYS THEN DOITTOIT
{$END}
```

$INCLUDE

Allows the compiler to switch processing from the current source to the file named. When the end of the file that was included is reached, the compiler switches back to the original source and continues compilation. Resumption of compilation in the original source file begins with the line of source text that follows the line in which the $INCLUDE occurred. Therefore, the $INCLUDE metacommand should always be last on a line.

$INCONST

Allows you to enter the values of the constants (such as those used in $IFs) at compile time, rather than editing the source. This is useful when you use metaconditionals to compile a version of a source for a particular environment, customer, etc. Compilation may be either interactive or batch oriented. For example, the metacommand $INCONST:YEAR produces the following prompt for the constant YEAR:

```
Inconst:  YEAR =
```

You need only give a response like:

```
Inconst:  YEAR = 1983
```

The response is presumed to be of type WORD. The effect is to declare a constant identifier named YEAR with the value 1983. This interactive setting of the constant YEAR is equivalent to the constant declaration:

```
CONST YEAR = 1983;
```

$MESSAGE

Allows you to send messages to your terminal during compilation. This is particularly useful if you use metaconditionals extensively, for example, and need to know which version of a program is being compiled.

Example of the $MESSAGE metacommand:

```
{$MESSAGE:'Message on terminal screen!'}
```

$PUSH and $POP

Allow you to create a meta-environment you can store with $PUSH and invoke with $POP. $PUSH and $POP are useful in $INCLUDE files for saving and restoring the metacommands in the main source file.

LISTING FILE CONTROL

The following metacommands allow you to format the listing file as you wish:

Metacommand	Description
$LINESIZE:<n>	Sets width of listing. Default is 131.
$LIST+	Turns on or off source listing. Errors are always listed.
$OCODE+	Turns on disassembled object code listing.
$PAGE+	Skips to next page. Line number is not reset.
$PAGE:<n>	Sets page number for next page (does not skip to next page).
$PAGEIF:<n>	Skips to next page if less than n lines left on current page.
$PAGESIZE:<n>	Sets length of listing in lines.

	Default is 55.
$SKIP:<n>	Skips n lines or to end of page.
$SUBTITLE:'<text>'	Sets page subtitle.
$SYMTAB+	Sends symbol table to listing file.
$TITLE:'<text>'	Sets page title.

$LINESIZE:<n>

Sets the maximum length of lines in the listing file. This value normally defaults to 131.

$LIST+

Turns on the source listing. Except for $LIST-, metacommands themselves appear in the listing. The format of the listing file is described in chapter 14, Compiling, linking, and Executing programs.

$OCODE+

Turns on the symbolic listing of the generated code to the object listing file. Although the format varies with the target code generator, it generally looks like an assembly listing, with code addresses and operation mnemonics. In many cases, the identifiers for procedure, function, and static variables are truncated in the object listing file.

$PAGE+

Forces a new page in the source listing. The page number of the listing file is automatically incremented.

$PAGE:<n>

Sets the page number of the next page of the source listing. $PAGE:<n> does not force a new page in the listing file.

$PAGEIF:<n>

Conditionally performs $PAGE+, if the current line number

of the source file plus n is less than or equal to the current page size.

$PAGESIZE:<n>

Sets the maximum size of a page in the source listing. The default is 55 lines per page.

$SKIP:<n>

Skips n lines or to the end of the page in the source listing.

$SUBTITLE:'<subtitle>'

Sets the name of a subtitle that appears beneath the title at the top of each page of the source listing.

$SYMTAB+

If on at the end of a procedure, function, or compiland, sends information about its variables to the listing file (for example, see lines 14 and 17 in the sample listing file in chapter 14, Compiling, Linking, and Executing Programs). The left columns contain the following:

1. the offset to the variable from the frame pointer (for variables in procedures and functions)
2. the offset to the variable in the fixed memory area (for main program and STATIC variables)
3. the length of the variable

A leading plus or minus sign indicates a frame offset. Note that this offset is to the lowest address used by the variable.

The first line of the $SYMTAB listing contains the offset to the return address, from the top of the frame (zero for the main program), and the length of the frame, from the frame pointer to the end including front end temporary variables. Code generator temporary variables are not included.

For functions, the second line contains the offset, length, and type of the value returned by the functions. The remaining lines list the variables, including their

type and attribute keywords, as shown below:

Keyword	Meaning
Public	Has the PUBLIC attribute
Extern	Has the EXTERN attribute
Origin	Has the ORIGIN attribute
Static	Has the STATIC attribute
Const	Has the READONLY attribute
Value	Occurs in a VALUE section
ValueP	Is a value parameter
VarP	Is a VAR or CONST parameter
VarsP	Is a VARS or CONSTS parameter
ProcP	Is a procedural parameter
Segmen	Uses segmented addressing
Regist	Parameter passed in register

$TITLE:'<title>'

Sets the name of a title that appears at the top of each page of the source listing.

For information on Listing File format see, Chapter 14, Compiling, linking, and Executing Programs.

CHAPTER 5
IDENTIFIERS AND CONSTANTS

CONTENTS

IDENTIFIERS

An identifier consists of a single letter followed by additional letters or digits, or underscore (_). Identifiers denote constants, variables, procedures, functions, programs and tag fields in records. Some features also use identifiers, such as super arrays, types, modules, units and statement labels.

Identifiers can be of any length, but must fit on a line. Only the first 31 characters are significant. An identifier longer than the significant length causes the compiler to generate a warning and not a fatal error.

The identifiers used for a program, module, or unit are passed to the linker, as well as identifiers with PUBLIC or EXTERN attribute.

The disassembled object code listing and debugger symbol table may truncate variable and procedural identifiers to 6 characters. Using identifiers of 7 or fewer characters saves time during compilation.

All external identifiers used internally by the runtime system are 4 alphabetic characters followed by "QQ". This form should be avoided when using new external names.

The Scope of Identifiers

An identifier is defined for the duration of the procedure, function, program, module, implementation, or interface in which you declare it. This holds true for any nested procedures or functions. An identifier's association must be unique within its scope; that is, it must not name more than one thing at a time.

A nested procedure or function can redefine an identifier only if the identifier has not already been used in it. However, the compiler does not identify such redefinition as an error, but will generally use the first definition until the second occurs. A special exception for reference types is discussed in chapter 6, Data Types.

Predeclared Identifiers

This category includes the identifiers for predeclared types, super array types, constants, file variables, functions, and procedures. You can use them freely, without declaring them. However, they differ from reserved words in that you may redefine them whenever you wish. At the standard level, the following identifiers are predeclared:

ABS	FALSE	OUTPUT	ROUND
ARCTAN	FLOAT	PAGE	SIN
BOOLEAN	GET	PACK	SQR
CHAR	INPUT	PRED	SQRT
CHR	INTEGER	PUT	SUCC
COS	LN	READ	TEXT
DISPOSE	MAXINT	READLN	TRUE
EOF	NEW	REAL	TRUNC
EOLN	ODD	RESET	UNPACK
EXP	ORD	REWRITE	WRITE
			WRITELN

The following identifiers are available at the extended and system levels:

1. String intrinsics

CONCAT	INSERT
COPYLST	POSITN
COPYSTR	SCANEQ
DELETE	SCANNE

2. Extended level intrinsics

ABORT	HIBYTE
BYWORD	LOBYTE
DECODE	LOWER
ENCODE	RESULT
EVAL	SIZEOF
	UPPER

3. System level intrinsics

FILLC	MOVESL
FILLSC	MOVESR
MOVEL	RETYPE
MOVER	

4. Extended level I/O

ASSIGN	READFN
CLOSE	READSET
DIRECT	SEEK
DISCARD	SEQUENTIAL
FCBFQQ	TERMINAL
FILEMODES	

5. INTEGER4 type

BYLONG	LOWORD

```
FLOATLONG    MAXINT4
HIWORD       ROUNDLONG
INTEGER4     TRUNCLONG
```

6. Super array type

```
LSTRING
NULL
STRING
```

7. WORD type

```
MAXWORD
WORD
WRD
```

8. Miscellaneous

```
ADRMEM       INTEGER2
ADSMEM       REAL4
BYTE         REAL8
INTEGER1     SINT
```

CONSTANTS

A constant is a value that is known before a program starts and that will not change as the program progresses. Examples of constants include the number of days in the week, your birthdate, the name of your dog, or the phases of the moon.

A constant may be given an identifier, but you cannot alter the value associated with that identifier during the execution of the program. Each constant implicitly belongs to some category of data:

1. Numeric constants are one of the several number types: REAL, INTEGER, WORD, or INTEGER4.

2. Character constants are strings of characters enclosed in single quotation marks and are called "string literals" in Pascal.

3. Structured constants include constant arrays, records, and typed sets.

Constant expressions allow you to compute a constant based on the values of previously declared constants in expressions. The identifiers defined in an enumerated type are constants of that

type and cannot be used directly with numeric (or string) constant expressions. These identifiers can be used with the ORD, WRD, or CHR functions (e.g., ORD (BLUE)).

TRUE and FALSE are predeclared constants of type BOOLEAN and can be redeclared. NIL is a constant of any pointer type; however, because it is a reserved word, you may not redefine it. Also, the null set is a constant of any set type.

Numeric statement labels have nothing to do with numeric constants; you may not use a constant identifier or expression as a label. Internally, all constants are limited in length to a maximum of 255 bytes.

Constant Identifiers

A constant identifier introduces the identifier as a synonym for the constant. You should put these declarations in the CONST section of a compiland, procedure, or function.

The general form of a constant identifier declaration is the identifier followed by an equal sign and the constant value. The following program fragment includes three statements that identify constants (beginning after the word "CONST"):

```
PROGRAM DEMO (INPUT, OUTPUT);
CONST DAYSINYEAR = 365;
      DAYSINWEEK = 7;
      NAMEOFPLANET = 'EARTH';
```

In this example, the numbers 365 and 7 are numeric constants; 'EARTH' is a string literal constant and must be enclosed in single quotation marks. ISO Pascal defines a strict order in which to set out the declarations in the declaration section of a program:

```
CONST MAX = 1Ø;
TYPE NAME = PACKED ARRAY [1..MAX] OF CHAR;
VAR FIRST : NAME;
```

The extended level of Pascal relaxes this order and, in fact, allows more than one instance of each kind of declaration:

```
TYPE  COMPLEX = RECORD
                  R, I : REAL;
      END;
CONST
      PII = COMPLEX (3.1416, ØØ);
VAR
      PIX : COMPLEX;
```

```
TYPE
      IVEC = ARRAY [1..3] OF COMPLEX;
CONST
      PIVEC = IVEC (PII, PII, COMPLEX (Ø.Ø, 1.Ø));
```

Numeric Constants

Numeric constants are irreducible numbers such as 45, 12.3, and 9E12. The notation of a numeric constant generally indicates its type: REAL, INTEGER, WORD, or INTEGER4. Numbers can have a leading plus sign (+) or minus sign (-), except when the numbers are within expressions. Therefore:

```
ALPHA := +1Ø       {Valid}

ALPHA + -1Ø        {Invalid}
```

The compiler truncates any number that exceeds a certain maximum number of characters and gives a warning when this occurs. The maximum length of constants (31) is the same as the maximum length of identifiers.

The syntax for numeric constants applies not only to the actual text of programs, but also to the content of textfiles read by a program.

Examples of numeric constants:

```
 123                 Ø.17
+12.345              ØØ7
-1.7E-1Ø            -26.Ø
 17E+3               26.ØE12
-17E3                1E1
```

Numeric constants can appear in any of the following:

1. CONST sections
2. expressions
3. type clauses
4. set constants
5. structured constants
6. CASE statement CASE constants
7. variant record tag values

REAL Constants

The type of a number is REAL if the number includes a decimal point or exponent. This provides about seven digits of precision, with a maximum value of about 1.7Ø1411E38. There is, however, a distinction between REAL values and REAL constants. The REAL constant range may be a subset of the REAL value range. The REAL numeric constants must be greater than or equal to 1.ØE-38 and less than 1.ØE+38.

The compiler issues a warning if there is not at least one digit on each side of a decimal point. A REAL number starting or ending with a decimal point may be misleading. For example, because left parenthesis-period substitutes for left square bracket, and right parenthesis-period for right square bracket, the following:

```
(.1+2.)
```

is interpreted as:

```
[1+2]
```

Scientific notation in REAL numbers (as in 1.23E-6 or 4E7) is supported. The decimal point and exponent sign are optional when an exponent is given. Both the uppercase "E" and the lowercase "e" are allowed in REAL numbers. "D" and "d" are also allowed to indicate an exponent. This provides compatibility with other languages.

All real constants are stored in REAL8 (double precision) format. If you require a single precision REAL4 constant, declare a REAL4 variable and give it your real constant value in a VALUE section.

INTEGER, WORD, and INTEGER4 Constants

The type of a non-REAL numeric constant is INTEGER, WORD, or INTEGER4. The constants of each of these types can assume the following range of values:

Type	**Range of Values (minimum/maximum)**	**Predeclared Constant**
INTEGER	-MAXINT to MAXINT	MAXINT=32767
WORD	Ø to MAXWORD	MAXWORD=65535

INTEGER4 -MAXINT4 to MAXINT4 MAXINT4=2147483647

MAXINT, MAXWORD, and MAXINT4 are all predeclared constant identifiers. One of three things happens when you declare a numeric constant identifier:

1. A constant identifier from -MAXINT to MAXINT becomes an INTEGER.

2. A constant identifier from MAXINT+1 to MAXWORD becomes a WORD.

3. A constant identifier from -MAXINT4 to -MAXINT-1 or MAXWORD+1 to MAXINT4 becomes an INTEGER4.

However, any INTEGER type constant (including constant expressions and values from -32767 to -1) automatically changes to type WORD if necessary; if the INTEGER value is negative, 65536 is added to it and the underlying 16-bit value is not changed.

For example, you can declare a subrange of type WORD as WRD(Ø)..127; the upper bound of 127 is automatically given the type WORD. The reverse is not true; constants of type WORD are not automatically changed to type INTEGER.

The ORD and WRD functions also change the type of an ordinal constant to INTEGER or WORD. Also, any INTEGER or WORD constant automatically changes to type INTEGER4 if necessary, but the reverse is not true.

The following are examples of constant conversions:

Constant	**Assumed Type**
Ø	INTEGER could become WORD or INTEGER4
-32768	INTEGER4 only
32768	WORD could become INTEGER4
Ø..2ØØØØ	INTEGER subrange
Ø..5ØØØØ	WORD subrange
Ø..8ØØØØ	Invalid: no INTEGER4 subranges
-1..5ØØØØ	Invalid: becomes 65535..5ØØØØ (i.e., -1 is treated as 65536)

Nondecimal Numbering

Pascal supports not only decimal number notation, but also numbers in hexadecimal, octal, binary, or other base numbering (where the base can range from 2 to 36). The number sign (#) acts as a radix separator.

Examples of numbers in nondecimal notation:

```
16#FFØ2
1Ø#987
 8#776
 2#111ØØ
```

Leading zeros are recognized in the radix, so a number like ØØ8#147 is permitted. In hexadecimal notation, upper or lowercase letters A through F are permitted. A nondecimal constant without the radix (such as #44) is assumed to be hexadecimal. Nondecimal notation does not imply a WORD constant and may be used for INTEGER, WORD, or INTEGER4 constants. You must not use nondecimal notation for REAL constants or numeric statement labels.

Character Strings

In pascal sequences of characters enclosed in single quotation marks are called "string literals" to distinguish them from string constants, which may be expressions, or values of the STRING type.

A string constant contains from 1 to 255 characters. A string constant longer than one character is of type PACKED ARRAY [1..n] OF CHAR, also known as the type STRING (n). A string constant that contains just one character is of type CHAR. However, the type changes from CHAR to PACKED ARRAY [1..1] OF CHAR (e.g., STRING (1)) if necessary. For example, a constant ('A') of type CHAR could be assigned to a variable of type STRING (1).

A literal apostrophe (single quotation mark) is represented by two adjacent single quotation marks (e.g., 'DON''T GO'). The null string ('') is not permitted. A string literal must fit on a line. The compiler recognizes string literals enclosed in double quotations marks (") or accent marks (`), instead of single quotation marks, but issues a warning message when it encounters them.

You can have string constants made up of concatenations of other string constants, including string constant identifiers, the CHR

() function, and structured constants of type STRING. This is useful for representing string constants that are longer than a line or that contain nonprinting characters. For example:

```
'THIS IS UNDERLINED' * CHR(13) * STRING (DO 18 OF '_')
```

The LSTRING feature adds the super array type LSTRING. LSTRING is similar to PACKED ARRAY [Ø..n] OF CHAR, except that element Ø contains the length of the string, which can vary from Ø to a maximum of 255. Note that, if necessary, a constant of type STRING (n) or CHAR changes automatically to type LSTRING. See chapter 6, Data Types for a discussion of LSTRINGs.

NULL is a predeclared constant for the null LSTRING, with the element Ø (the only element) equal to CHR (Ø). NULL cannot be concatenated, since it is not of type STRING. It is the only constant of type LSTRING.

Examples of string literal declarations:

```
NAME = 'John Jacob';      {a valid string literal}
LETTER = 'Z';             {LETTER is of type CHAR}
QUOTED_QUOTE = '''';      {Quotes quote}
NULL_STRING = NULL;       {Invalid}
NULL_STRING = '';         {Invalid}
DOUBLE = "OK";            {generates a warning}
```

Structured Constants

ISO Pascal permits only the numeric and string constants already mentioned, the pointer constant value NIL, and untyped constant sets. With this Pascal, you may use constant arrays, records, and typed sets. Structured constants can be used anywhere a structured value is allowed, in expressions as well as in CONST and VALUE sections.

1. An array constant consists of a type identifier followed by a list of constant values in parentheses separated by commas.

   Example of an array constant:

   ```
   TYPE   VECT_TYPE = ARRAY [-2..2] OF INTEGER;
   CONST  VECT = VECT_TYPE (5, 4, 3, 2, 1);
   VAR    A : VECT_TYPE;
   VALUE  A := VECT;
   ```

2. A record constant consists of a type identifier followed by a list of constant values in parentheses separated by commas.

   Example of a record constant:

```
TYPE REC_TYPE = RECORD
       A, B: BYTE;
       C, D: CHAR;
     END;
CONST RECR = REC_TYPE (#20, 0, 'A', CHR (20));
VAR   FOO : REC_TYPE;
VALUE FOO := RECR;
```

3. A set constant consists of an optional set type identifier followed by set constant elements in square brackets. Set constant elements are separated by commas. A set constant element is either an ordinal constant, or two ordinal constants separated by two dots to indicate a range of constant values.

   Example of a set constant:

```
TYPE  COLOR_TYPE = SET OF (RED, BLUE, WHITE, GRAY,
                           GOLD);
CONST SETC = COLOR_TYPE [RED, WHITE .. GOLD];
VAR   RAINBOW : COLOR_TYPE;
VALUE RAINBOW := SETC;
```

A constant within a structured array or record constant must have a type that can be assigned to the corresponding component type. For records with variants, the value of a constant element corresponding to a tag field selects a variant, even if the tag field is empty. The number of constant elements must equal the number of components in the structure, except for super array type structured constants. Nested structured constants are permitted.

An array or record constant nested within another structured constant must still have the preceding type identifier. For this reason, a super array constant can have only one dimension (see chapter 6, Data Types, for a discussion of super arrays). The size of the representation of a structured constant must be from 1 to 255 bytes. If this 255-byte limit is a problem, declare a structured variable with the READONLY attribute, and initialize its components in a VALUE section.

Example of a complex structured constant:

```
TYPE R3 = ARRAY [1..3] OF REAL;
TYPE SAMPLE = RECORD I: INTEGER;
                     A: R3;
```

```
                        CASE BOOLEAN OF
                        TRUE: (S: SET OF 'A'..'Z';
                               P: ^ SAMPLE);
                        FALSE:(X: INTEGER);
                END;
CONST SAMP_CONST= SAMPLE (27, R3 (1.4, 1.4, 1.4),
                TRUE, ['A','E','I'], NIL);
```

Constant elements can be repeated with the phrase DO <n> OF <constant>, so the previous example could have included "DO 3 OF 1.4" instead of "1.4, 1.4, 1.4".

Pascal does not support set constant expressions, such as ['_'] + LETTERS, or file constant expressions. The constant 'ABC' of type STRING (3) is equivalent to the structured constant STRING ('A', 'B', 'C'). LSTRING structured constants are not permitted; use the corresponding STRING constants instead.

Structured constants (and other structured values, such as variables and values returned from functions) can be passed by reference using CONST parameters. For more information, see chapter 10, Procedures and Functions.

There are two kinds of set constants: one with an explicit type, as in CHARSET ['A'..'Z'], and one with an unknown type, as in [20..40]. You may use either in an expression or to define the value of a constant identifier. Set constants with an explicit type may also be passed as a reference (CONST) parameter. Sets of unknown type are unpacked, but the type changes to PACKED if necessary.

Constant Expressions

Constant expressions allow you to compute constants based on the values of previously declared constants in expressions. Constant expressions can also occur within program statements.

Example of a constant expression declaration:

```
CONST HEIGHT_OF_LADDER = 6;
      HEIGHT_OF_MAN    = 6;
      REACH  = HEIGHT_OF_LADDER + HEIGHT_OF_MAN;
```

Because a constant expression may contain only constants that you have declared earlier, the following is invalid:

```
CONST MAX = A + B;
      A   = 10;
      B   = 20;
```

Certain functions may be used within constant expressions. For example:

```
CONST A = LOBYTE (-23) DIV 23;
      B = HIBYTE (-A);
```

Listed below are functions and operators that may be used with REAL, INTEGER, WORD, and other ordinal constants, such as enumerated and subrange constants.

Type of Operand	Functions and Operators
REAL, INTEGER	Unary plus (+) Unary minus (-)
INTEGER, WORD	+ DIV OR HIBYTE() - MOD NOT LOBYTE() * AND XOR BYWORD()
Ordinal types	< <= CHR() LOWER() > >= ORD() UPPER() = <> WRD()
Boolean	AND NOT OR
ARRAY	LOWER() UPPER()
Any type	SIZEOF() RETYPE()

Examples of constant expressions:

```
CONST FOO = (100 + ORD('X')) * 8#100 + ORD('Y');
      MAXSIZE = 80;
      X = (MAXSIZE > 80) OR (IN_TYPE = PAPERTAPE);
      {X is a BOOLEAN constant}
```

In addition to the operators shown above for numeric constants, you may use the string concatenation operator (*) with string constants, as follows:

```
CONST A = 'abcdef';
      M = CHR (109);          {CHR is allowed}
      ATOM = A * 'ghijkl' * M;{ATOM = 'abcdefghijklm'}
```

These constants can span more than one line, but are still limited to the 255 character maximum. These string constant expressions are allowed wherever a string literal is allowed, except in metacommands.

CHAPTER 6
DATA TYPES

CONTENTS

A type is the set of values that a variable or value can have within a program. Types are either predeclared or declared explicitly. Types in Pascal fall into three broad categories: simple, structured, and reference types.

Simple Types	Ordinal types	
	INTEGER	-MAXINT..MAXINT
	WORD	Ø..MAXWORD
	CHAR	CHR(Ø)..CHR(255)
	BOOLEAN	(FALSE,TRUE)
	enumerated types	e.g., (RED,BLUE)
	subrange types	e.g., 1ØØ..5ØØØ
	REAL4, REAL8	
	INTEGER4	-MAXINT4..MAXINT4
Structured Types	ARRAY OF type	
	General (OF any type)	
	SUPER ARRAY (OF type)	
	STRING (n)	[1..n] of CHAR
	LSTRING (n)	[Ø..n] of CHAR
	RECORD	
	SET OF type	
	FILE OF	
	general (binary) files	
	TEXT	Like FILE OF CHAR
Reference Types	Pointer Types	e.g., ^TREETIP
	ADR OF type	Relative address
	ADS OF type	Segmented address
Procedural and Functional Types		Only as parameter type

The type declaration associates an identifier with a type of value. You declare types in the TYPE section of a program, procedure, function, module, interface, or implementation (not in the heading of a procedure or function). A type declaration consists of an identifier followed by an equal sign and a type clause.

Examples of type definitions:

```
TYPE LINE = STRING (8Ø);
     PAGE = RECORD
         PAGENUM : 1..499;
         LINES : ARRAY [1..6Ø] OF LINE;
```

```
        FACE : (LEFT, RIGHT);
        NEXTPAGE : ^PAGE;
        END;
```

After declaring the types, you can declare variables of the types just defined in the VAR section of a program, procedure, function, module, or interface, or in the heading of a procedure or function. The following sample VAR section declares variables of the types in the preceding sample TYPE section:

```
VAR PARAGRAPH : LINE;
    BOOK : PAGE;
```

SIMPLE DATA TYPES

The simple data types are organized as follows:

1. ordinal types
2. REAL
3. INTEGER4

Ordinal Types

Ordinal types are all finite and countable. They include the following simple types:

INTEGER
WORD
CHAR
BOOLEAN
enumerated types
subrange types

INTEGER4, though finite and countable, is not an ordinal type.

INTEGER

INTEGER values are a subset of the whole numbers and range from -MAXINT through Ø to MAXINT. MAXINT is the predeclared constant 32767 (i.e., 2^15 - 1). The value -32768 is not a valid INTEGER; the compiler uses it to check for uninitialized INTEGER and INTEGER subrange variables.

INTEGER is not a subrange of INTEGER4. If it were, signed expressions would have to be calculated using the INTEGER4 type and the result converted to INTEGER.

Expressions are always calculated using a base type, not a subrange type. INTEGER type constants may be changed internally to WORD type if necessary, but INTEGER variables are not. INTEGER values change to REAL or INTEGER4 in an expression, if necessary, but not to REAL. The ORD function converts a value of any ordinal type to an INTEGER type.

The predeclared type INTEGER2 is identical to INTEGER.

WORD

The WORD and INTEGER types are similar, differing chiefly in their range of values. Both are ordinal types. You can think of WORD values as either a group of 16 bits or as a subset of the whole numbers from Ø to MAXWORD (65535, i.e., 2^16 - 1). The WORD type is is useful in several ways:

1. to express values in the range from 32768 to 65535
2. to operate on machine addresses
3. to perform primitive machine operations, such as word ANDing and word shifting, without using the INTEGER type and running into the -32768 value

Unlike INTEGERs, all WORDs are nonnegative values. The WRD function changes any ordinal type value to WORD type. Like INTEGER values, WORD values in an expression are converted to the INTEGER4 type, if necessary. Having both an INTEGER and a WORD type permits mapping of 16-bit quantities in either of two ways:

1. as a signed value ranging from -32767 to +32767
2. as a positive value ranging from Ø to 65535.

WORD and INTEGER values not assignment compatible. However, you must not mix WORD and INTEGER values in an expression (although doing so generates a warning rather than an error message).

CHAR

CHAR values are 8-bit ASCII values. CHAR is an ordinal type. All 256-byte values are included in the type CHAR. In addition, SET OF CHAR is supported. Relational comparisons use the ASCII collating sequence.

The CHR function changes any ordinal type value to CHAR type, as long as ORD of the value is in the range from Ø to 255. See Appendix I, "ASCII Character Codes," for a complete listing of the ASCII character set.

BOOLEAN

BOOLEAN is an ordinal type with only two (predeclared) values: FALSE and TRUE. The BOOLEAN type is a special case of an enumerated type, where ORD (FALSE) is Ø and ORD (TRUE) is 1. This means that FALSE < TRUE.

You may redefine the identifiers BOOLEAN, FALSE, and TRUE, but the compiler implicitly uses the former type in Boolean expressions and in IF, REPEAT, and WHILE statements.

There is no function that changes ordinal type values to the BOOLEAN type. However, you can achieve this effect with the ODD function for INTEGER and WORD values, or the expression:

```
ORD (value) <> Ø
```

Enumerated Types

An enumerated type defines an ordered set of values. These values are constants and are enumerated by the identifiers that denote them.

Examples of enumerated type declarations:

```
FLAGCOLOR = (RED, WHITE, BLUE)
SUITS = (CLUB, DIAMOND, HEART, SPADE)
DOGS = (MAUDE, EMILY, BRENDAN)
```

Every enumerated type is also an ordinal type. Identifiers for all enumerated type constants must be unique within their declaration level.

The ORD function, can be used to change enumerated values into INTEGER values; the WRD function changes enumerated values into WORD values.

The RETYPE function, can be used to change INTEGER or WORD values to an enumerated type. For example:

```
IF RETYPE (COLOR, I) = BLUE THEN WRITELN ('TRUE BLUE')
```

The values obtained by applying the ORD function to the constants of an enumerated type always begin with zero. Thus, the values obtained for the type FLAGCOLOR, from the example above, are as follows:

```
ORD (RED)   = Ø
ORD (WHITE) = 1
ORD (BLUE)  = 2
```

Since enumerated types are ordered, comparisons like RED < GREEN can be useful. At times, access to the lowest and highest values of the enumerated type is useful with the LOWER and UPPER functions, as in the following example:

```
VAR TINT: COLOR;
FOR TINT := LOWER (TINT) TO UPPER (TINT)
   DO PAINT (TINT)
```

Subrange Types

A subrange type is a subset of an ordinal type. The type from which the subset is taken is called the "host" type. Therefore, all subrange types are also ordinal types.

You can define a subrange type by giving the lower and upper bound of the subrange (in that order). The lower bound must not be greater than the upper bound, but the bounds may be equal. The subrange type is frequently used as the index type of an array bound or as the base type of a set.

Examples of subranges along with their host ordinal type:

```
INTEGER            100..200
WORD               WRD(1)..9
CHAR               'A'..'Z'
enumerated type    RED..YELLOW
```

Three subrange types are predeclared:

1. BYTE = WRD(0)..255; {8-bit WORD subrange}

2. SINT = -127..127; {8-bit INTEGER subrange}

3. INTEGER1 = SINT

The BYTE type is particularly useful in machine-oriented applications. For example, the ADRMEM and ADSMEM types normally treat memory as an array of bytes. However, since the BYTE type is really a subrange of the WORD type, expressions with BYTE values are calculated using 16-bit instead of 8-bit arithmetic, if necessary.

In some cases (for example, an assignment of a BYTE expression to a BYTE variable when the $MATHCK switch is off), the compiler can optimize 16-bit arithmetic to 8-bit arithmetic. In general, using BYTE instead of WORD saves memory at the expense of BYTE-to-WORD conversions in expression calculations.

Real

REAL values are nonordinal values of a given range and precision. The Real formats have a 24-bit mantissa and an 8-bit exponent, giving about seven digits of precision and a maximum value of 1.7Ø1411E38.

Pascal includes expanded numeric data types for processing higher precision real (and integer) numbers. For reals, this includes support for single and double precision real numbers according to the IEEE floating-point standard.

Pascal provides three real types: REAL, REAL4, and REAL8. However, the type REAL is always identical to either REAL4 or REAL8. The choice is made with a metacommand, $REAL:n, where n is either 4 or 8. {$REAL:8} has the same effect as TYPE REAL = REAL8. The default type for REAL is normally REAL4, but may be changed.

The REAL4 type is in 32-bit IEEE format, and the REAL8 type is in 64-bit IEEE format. The IEEE standard format is as follows:

REAL4 Sign bit, 8-bit binary exponent with bias of 127, 23-bit mantissa

REAL8 Sign bit, 11-bit binary exponent with bias of 1Ø23, 52-bit mantissa

In both cases the mantissa has a "hidden" most significant bit (always one) and represents a number greater than or equal to 1.Ø but less than 2.Ø. An exponent of zero means a value of zero, and the maximum exponent means a value called NaN (not a number). Bytes are in "reverse" order; the lowest addressed byte is the least significant mantissa byte.

The REAL4 numeric range is barely seven significant digits (24 bits), with an exponent range of E-38 to E+38. The REAL8 numeric range includes over fifteen significant digits (53 bits), with an exponent range of E-3Ø6 to E+3Ø6 (a very large number!)

REAL literals are converted first to REAL8 format and then to REAL4 as necessary (for example, to be passed as a CONST parameter or to initialize a variable in a VALUE section). If you need actual REAL4 constants, you must declare them as REAL4 variables (by adding the READONLY attribute) and assign them a constant in a VALUE section.

Both REAL4 and REAL8 values are passed to intrinsic functions as reference (CONSTS) parameters, rather than as value parameters. The compiler accepts REAL expressions as CONSTS parameters; it will evaluate the expression, assign the result to a stack

temporary, and pass the address of the temporary, which is usually more efficient than passing the value itself (especially in the REAL8 case).

Integer4

Like INTEGER and WORD values, INTEGER4 values are a subset of the whole numbers. INTEGER4 values range from -MAXLONG to MAXLONG. MAXLONG is a predeclared constant with the value of 2,147,483,647 (i.e., 2^31 - 1). The value -2,147,487,648 (i.e., -2^31) is not a valid INTEGER4.

Unlike INTEGER and WORD, the INTEGER4 type is not considered an ordinal type. There are no INTEGER4 subranges and INTEGER4 cannot be an array index or the base type of a set. Also, INTEGER4 values cannot be used to control FOR and CASE statements.

Values of type INTEGER or WORD in an expression change automatically to INTEGER4 if the expression requires an intermediate value that is out of the range of either INTEGER or WORD. Values of type INTEGER4 do not change to REAL in an expression; you must explicitly use the FLOATLONG function to make the conversion.

STRUCTURED DATA TYPES

A structured type is composed of other types. The components of structured types are either simple types or other structured types. A structured type is characterized by the types of its components and by its structuring method. A structured type can occupy up to 65534 bytes of memory. The structured types in Pascal are the following:

```
ARRAY range OF type

SUPER ARRAY range OF type

   STRING  (n)

   LSTRING (n)

RECORD

SET OF <base-type>

FILE OF <type>
```

Because components of structures can be structured types themselves, you may have, for example, an array of arrays, a file of records containing sets, or a record containing a file and

another record.

Arrays

An array type is a structure that consists of a fixed number of components. All of the components are of the same type (called the "component type").

The elements of the array are designated by indices, which are values of the "index type" of the array. The index type must be an ordinal type: BOOLEAN, CHAR, INTEGER, WORD, subrange, or enumerated.

Arrays in Pascal are one dimensional, but since the component type can also be an array, n-dimensional arrays are supported as well.

Examples of type declarations for arrays:

```
TYPE
INT_ARRAY : ARRAY [1..10] OF INTEGER;
ARRAY_2D  : ARRAY [0..7] OF ARRAY [0..8] OF 0..9;
MORAL_RAY : ARRAY [PEOPLE] OF (GOOD, EVIL)
```

In the last declaration, PEOPLE is a subrange type, while GOOD and EVIL are enumerated constants. A short-hand notation available for n-dimensional arrays makes the following statement the same as the second example in the preceding paragraph:

```
ARRAY_2D :  ARRAY [0..7, 0..8] OF 0..9;
```

After declaring these arrays, you could assign to components of the arrays with statements such as these:

```
INT_ARRAY [10]  := 1234;
ARRAY_2D [0,99] := 999;
MORAL_RAY [Machiavelli] := EVIL;
```

All of an n-dimensional PACKED array is packed; therefore these statements are equivalent:

```
PACKED ARRAY [1..2, 3..4] OF REAL

PACKED ARRAY [1..2] OF PACKED ARRAY [3..4] OF REAL
```

Super Arrays

A super array is an example of "super type." A super type is like a set of types or like a function that returns a type. Super types in general, and super arrays in particular, are

features of this extended Pascal. The super array type has several important uses. You may use them for any of the following purposes:

1. To process strings.

   Both STRING and LSTRING are predeclared super array types. The LSTRING type handles variable length strings. STRING handles fixed-length strings and strings more than 255 characters long.

2. To dynamically allocate arrays of varying sizes.

   Otherwise such arrays would need a maximum possible size allocation.

3. As the formal parameter type in a procedure or function. Such a declaration makes the procedure or function usable for a set or class of types, rather than for just a single fixed-length type.

A super type identifier specifies the set of types represented by the super type. A later type declaration may declare a normal type identifier as a type "derived" from that class of types. This derived type is like any other type.

A super array type declaration is an array type declaration prefixed with the keyword SUPER. Every array upper bound is replaced with an asterisk, as follows:

```
TYPE VECTOR = SUPER ARRAY [1..*] OF REAL;
```

Following the preceding type declaration, you could declare the following variables:

```
VAR   ROW: VECTOR (10);
      COL: VECTOR (30);
      ROWP: ^VECTOR;
```

In this example, VECTOR is a super array type identifier. VECTOR (10) and VECTOR (30) are type designators denoting "derived types." ROW and COL are variables of types derived from VECTOR. ROWP is a pointer to the super array type VECTOR.

Super types allow only an array type with parametric upper bounds. A super type is a class of types and not a specific type. Thus, in the VAR section of a program, procedure, or function, you cannot declare the variables to be of a super type; you must declare them as variables of a type derived from the super type.

However, a formal reference parameter in a procedure or function can be given a super type; this allows the routine to operate on any of the possible derived types.

A pointer referent type can also be given a super type. This allows a pointer to refer to any of the possible derived types. A pointer referent to a super type allows "dynamic arrays." These arrays are allocated on the heap by passing their upper bound to the procedure NEW. See Chapter 11, "Available Procedures and Functions," for a description of the procedure NEW.

Example using the NEW procedure for dynamic allocation:

```
VAR STR_PNT: ^SUPER PACKED ARRAY [1..*] OF CHAR;
    VEC_PNT: ^SUPER ARRAY [Ø..*, Ø..*] OF REAL;
      .
      .
    NEW (STR_PNT, 12);
    NEW (VEC_PNT, 9, 99);
```

An actual parameter in a procedure or function can be of a super type rather than a derived type, but only if the parameter is a reference parameter or pointer referent. (These are the only kinds of variables that can be of a super rather than a derived type.)

Example of super arrays:

```
TYPE VECTOR = SUPER ARRAY [1..*] OF REAL;

VAR X: VECTOR (12); Y: VECTOR (24); Z: VECTOR (36);

FUNCTION SUM (VAR V: VECTOR): REAL;

VAR S: REAL; I: INTEGER;
BEGIN
  S := Ø;
  FOR I := 1 TO UPPER (V) DO S := S + V [I];
  SUM := S;
END;

BEGIN
  .
  .
  TOTAL := SUM (X) + SUM (Y) + SUM (Z);
  .
  .
END
```

The normal type rules for components of a super array type and for type designators that use a super array type allow components

to be assigned, compared, and passed as parameters.

The UPPER function returns the actual upper bound of a super array parameter or referent. The maximum upper bound of a type derived from a super array type is limited to the maximum value of the index type implied by the lower bound (e.g., MAXINT, MAXWORD). Two super array types are predeclared, STRING and LSTRING. The compiler directly supports STRING and LSTRING types in the following ways:

1. STRING and STRING assignment
2. STRING and STRING comparison
3. LSTRING and STRING READs
4. access to the length of a STRING with the UPPER function
5. access to maximum length of an LSTRING with the UPPER function
6. access to LSTRING length with STR.LEN and STR[Ø]

Strings

STRINGs are predeclared super arrays of characters:

```
TYPE STRING = SUPER PACKED ARRAY [1..*] OF CHAR;
```

A string literal such as 'abcdefg' automatically has the type STRING (n). The size of the array 'abcdefg' is 7; thus, the constant is of the STRING derived type, STRING (7).

Standard Pascal calls any packed array of characters with a lower bound of one a "string" and permits a few special operations on this type (such as comparison and writing) that you cannot do with other arrays.

The super array notation STRING (n) is identical to PACKED ARRAY [1..n] OF CHAR (n may range from 1 to MAXINT). There is no default for n, since STRING means the super array type itself and not a string with a default length.

The identifier STRING is for a super array, so you can only use it as a formal reference parameter type or pointer referent type. You cannot compare such a parameter or dereferenced pointer or assign it as a whole.

Any variable (or constant) with the super array type STRING, or one of the types CHAR or STRING (n) or PACKED ARRAY [1..n] OF CHAR, can be passed to a formal reference parameter of super

array type STRING. Furthermore, a variable of type LSTRING or LSTRING (n) can also be passed to a formal reference parameter of type STRING.

The standard level supports the assigning, comparing, and writing of STRINGs. The extended level permits reading STRINGs, including the super array type STRING and a derived type STRING (n). Reading a STRING causes input of characters until the end of a line or the end of the STRING is reached. If the end of the line is reached first, the rest of the STRING is filled with blanks. Writing a string writes all of its characters.

Any two variables or constants with the type PACKED ARRAY [1..n] OF CHAR or the type STRING (n) can be compared or assigned if the lengths are equal. However, since the length of a STRING super array type may vary, comparisons and assignments are not allowed.

For example:

```
PROCEDURE CANNOT_DO (VAR S : STRING);
VAR STR : STRING (10);
BEGIN
  R := S      {This assignment is illegal because the}
              {length of S may vary}
END;
```

The PACKED prefix in the declaration PACKED ARRAY [1..n] OF CHAR, as defined in the ISO standard, normally implies that a component cannot be passed as a reference parameter. At the extended level, this restriction does not apply.

The index type of a string is officially INTEGER, but WORD type values can also be used to index a STRING. A number of intrinsic procedures and functions for strings are discussed in Chapter 11, "Available Procedures and Functions." Many of the procedures and functions described work on STRINGs; some apply only to LSTRINGs.

Lstrings

The LSTRING feature allows variable-length strings. LSTRING (n) is predeclared as:

```
TYPE LSTRING = SUPER PACKED ARRAY [0..*] OF CHAR
```

However, a variable with the explicit type PACKED ARRAY [0..n] OF CHAR is not "identical" to the type LSTRING (n) even though they are structurally the same. There is no default for n; the range of n is from zero to 255. Characters in an LSTRING can be accessed with the usual array notation.

Internally, LSTRINGs contain a length (L), followed by a string of characters. The length is contained in element zero of the LSTRING and can vary from 0 to the upper bound. The length of an

LSTRING variable T can be accessed as T[Ø] with type CHAR, or as T.LEN with type BYTE. String constants of type CHAR or STRING (n) are changed automatically to type LSTRING.

The predeclared constant NULL is the empty string, LSTRING (Ø). NULL is the only constant with type LSTRING; there is no way to define other LSTRING constants. As with STRINGs, a CHAR component of an LSTRING can be passed as a reference parameter, and WORD and INTEGER values can be used to index an LSTRING.

Several operations work differently on LSTRINGs than on STRINGs. Any LSTRING can be assigned to any other LSTRING, so long as the current length of the right side is not greater than the maximum length of the left side. Similarly, an LSTRING can be passed as a value parameter to a procedure or function, so long as the current length of the actual parameter is not greater than the maximum length specified by the formal parameter.

If the $RANGECK is on, the compiler checks the assignment of LSTRINGs and the passing of LSTRING (n) parameters. The actual number of bytes assigned or passed is the minimum of the upper bounds of the LSTRINGs. Neither side in an LSTRING assignment can be a parameter of the super array type LSTRING; both must be types derived from it.

Examples of LSTRING assignments:

```
VAR A : LSTRING (19);   {Declaring the variables}
    B : LSTRING (14);
    C : LSTRING (6);
    .
    .
A := '19 character string';
B := '14 characters';
C := 'shorty';
A := B;     {This is legal, since the length of B
             is less than the maximum length of A.}

C := A;     {This is illegal, since length of A
             is greater than the maximum length of C.}
```

You may compare any two LSTRINGs, including super arraytype LSTRINGs (the only super array type comparison allowed). Reading an LSTRING variable causes input of characters, until the end of the current line or the end of the LSTRING, and sets the length to the number of characters read. Writing from an LSTRING writes the current length string.

Using Strings and Lstrings

This section describes the STRING and LSTRING operations directly supported by the compiler. See also Chapter 11, "Available Procedures and Functions," for descriptions of the following

string procedures and functions:

CONCAT	INSERT
COPYLST	POSITN
COPYSTR	SCANEQ
DELETE	SCANNE

The procedures FILLC, FILLSC, MOVEL, MOVESL, MOVER, and MOVESR also operate on strings. The compiler supports STRINGs and LSTRINGs directly in the following ways:

1. Assignment

You may assign any LSTRING value to any LSTRING variable, as long as the maximum length of the target variable is greater than or equal to the current length of the source value and neither is the super array type LSTRING. If the maximum length of the target is less than the current length of the source, only the target length is assigned, and a runtime error occurs if the range checking switch is on. You may assign a STRING value to a STRING variable, as long as the length of both sides is the same and neither side is the super array type STRING. Passing either STRING or LSTRING as a value parameter is much like making an assignment.

2. Comparison

The LSTRING operators < <= > >= <> = use the length byte for string comparisons; the operands may be of different lengths. Two strings must be the same length to be considered equal. If two strings of different lengths are equal up to the length of the shorter one, the shorter is considered less than the longer one. The operands can be of the super array type LSTRING. For STRINGs, the same relational operators are available, but the lengths must be the same and operands of the super array type STRINGs are not allowed.

3. READs and WRITEs

READ LSTRING reads until the LSTRING is filled or until the end-of-line is found. The current length is set to the number of characters read. WRITE LSTRING uses the current length. See also READSET (described in Chapter 12, "File-Oriented Procedures and Functions"), which reads into an LSTRING as long as input characters are in a given SET OF CHAR. READ STRING pads with spaces if the line is shorter than the STRING. WRITE STRING writes all the characters in the string. Both READ and WRITE permit the super array types STRING and LSTRING, as well as their derived types.

4. Length access

You can access the current length of an LSTRING variable T with T.LEN, which n is of type BYTE, or with T[Ø], which is of type CHAR. This notation can assign a new length, as well as determine the current length. The UPPER function will find the maximum length of an LSTRING or the length of a STRING. This is especially useful for finding the upper bound of a super array reference parameter or pointer referent.

You cannot assign or compare mixed STRINGs and LSTRINGs, unless the STRING is constant. You can assign STRINGs to LSTRINGs, or vice versa, with one of the move routines or with the COPYSTR and COPYLST procedures. Since constants of type STRING or CHAR change automatically to type LSTRING if necessary, LSTRING constants are considered normal STRING constants. NULL (the zero length LSTRING) is the only explicit LSTRING constant.

A "special transformation" lets you pass an actual LSTRING parameter to a formal reference parameter of type STRING. The length of the formal STRING is the actual length of the LSTRING. Therefore, if LSTR (in the following example) is of type LSTRING (n) or LSTRING, it can be passed to a procedure or function with a formal reference parameter of type STRING For example:

```
VAR LSTR : LSTRING (1Ø);
.
.
PROCEDURE TIE_STRING (VAR STR : STRING);
.
.
TIE_STRING (LSTR);
```

In this case, UPPER (STR) is equivalent to LSTR.LEN. Procedures and functions with reference parameters of super type STRING can operate equally well on STRINGs and LSTRINGs. The only reason to declare a parameter of type LSTRING is when the length must be changed. Normally, an LSTRING is either a VAR or a VARS parameter in a procedure or function, since a CONST or CONSTS parameter of type LSTRING cannot be changed.

Records

The record type is a structure consisting of a fixed number of components, usually of different types. Each component of a record type is called a field. The definition of a record type specifies the type and an identifier for each field within the record. The field values associated with field identifiers are accessible with a field designator or with the WITH statement.

For example, you could declare the following record type:

```
TYPE LP = RECORD
          TITLE : LSTRING (100);
          ARTIST : LSTRING (100);
          PLASTIC : ARRAY
            [1..SONG_NUMBER] OF SONG_TITLE
          END
```

You could then declare a variable of the type LP, as follows:

```
VAR BEATLES_1 :  LP;
```

A component of the record could be accessed either with the field designator or the WITH statement

```
BEATLES_1.TITLE := 'Meet The Beatles';
WITH BEATLES_1 DO
   PLASTIC[1] := 'I Wanna Hold Your Hand'
```

Variant Records

A record may have several "variants," in which case a certain field called the "tag field" indicates which variant to use. The tag field may or may not have an identifier and storage in the record. Some operations, such as the NEW and DISPOSE procedures and the SIZEOF function, can specify a tag value even if the tag is not stored as part of the record.

Examples of variant records:

```
TYPE OBJECT = RECORD
       X, Y: REAL;
       CASE S: SHAPE OF
         SQUARE: (SIZE, ANGLE: REAL);
         CIRCLE: (DIAMETER: REAL)
     END;

     FOO = RECORD
       CASE BOOLEAN OF
         TRUE: (I, J: INTEGER);
         FALSE: (CASE COLOR OF
                 BLUE: (X: REAL);
                 RED:  (Y: INTEGER4));
     END;
```

Only one variant part per record is allowed; it must be the last field of the record. However, this variant part can also have a variant (and so on, to any level). All field identifiers in a given record type must be unique, even in different variants. For example, after declaring the record types above, you could

create and then assign to the variables as shown in the following example:

```
VAR O, P : OBJECT;
    F, G : FOO;

BEGIN
  O.DIAMETER := 12.34;  {CASE of CIRCLE}
  P.SIZE := 1.2;        {CASE of SQUARE}
  F.I := 1; F.J := 2;   {CASE of TRUE}
  G.X := 123.45;        {CASE of FALSE and BLUE}
  G.Y := 678999         {CASE of FALSE and RED;}
                        {this over writes G.X.}
END;
```

Variant records interact with extended Pascal's features in two ways:

1. Declaring a variant that contains a file is not safe; any change to the file's data using a field in another variant may lead to I/O errors, even if the file is closed. In the following example, any use of R will lead to errors in F:

```
RECORD CASE BOOLEAN OF
    TRUE : (F: FILE OF REAL);
    FALSE : (R:ARRAY [1..100] OF REAL);
    END;
```

2. Giving initial data to several overlapping variants in a variable in a VALUE section could have unpredictable results. In the following example, the initial value of LAP is uncertain:

```
VAR LAP : RECORD CASE BOOLEAN OF
    TRUE : (I: INTEGER4);
    FALSE : (R: REAL);
    END;
VALUE LAP.I := 10; LAP.R := 1.5;
```

Explicit Field Offsets

You can assign explicit byte offsets to the fields in a record. This system level feature can be useful for interfacing to software in other languages, since control block formats may not conform to the usual field allocation method. However, because

it also permits unsafe operations, such as overlapping fields and word values at odd byte boundaries, it is not recommended unless the interface is necessary. The offset is enclosed in brackets; the number is the byte offset to the start of the field.

Example showing assignment of explicit byte offsets:

```
TYPE CPM = RECORD
             NDRIVE [ØØ]: BYTE;
             FILENM [Ø1]: STRING (8);
             FILEXT [Ø9]: STRING (3);
             EXTENT [12]: BYTE;
             CPMRES [13]: STRING (2Ø);
             RECNUM [33]: WORD;
             RECOVF [35]: BYTE;
           END;

     OVERLAP = RECORD
             BYTEAR [ØØ]: ARRAY [Ø..7] OF BYTE;
             WORDAR [ØØ]: ARRAY [Ø..3] OF WORD;
             BITSAR [ØØ]: SET OF Ø..63;
           END;
```

If you give any field an offset, give offsets to all fields. For any offset that you omit, the compiler picks an arbitrary value. Although the compiler will process a declaration that includes both offsets and variant fields, you should use only one or the other in a given program.

Although you can completely control field overlap with explicit offsets, variants provide the long forms of the procedures NEW, DISPOSE, and SIZEOF. If you want to allocate different length records, use the RETYPE and GETHQQ procedures, instead of variants and the long form of NEW. For example:

```
CPMPV := RETYPE (CPMP, GETHQQ (36));
```

The compiler does support structured constants for record types with explicit offsets. Internally, odd length fields greater than one are rounded to the next even length. For example:

```
ODDR = RECORD
         F1[ØØ] : STRING (3);
         F2[Ø3] : CHAR
       END;
```

In this example, field F1 is four bytes long, so an assignment to F1 over writes F2. In such a record, all odd length fields must be assigned first.

Sets

A set type defines the range of values that a set may assume. This range of assumable values is the "power set" of the base type you specify in the type definition. The power set is the set of all possible sets that could be composed from an ordinal base type. The null set, [], is a member of every set.

Suppose you declare the following set types:

```
TYPE HUES = SET OF COLOR;
     CAPS = SET OF 'A'..'Z';
     MATTER = SET OF (ANIMAL, VEGETABLE, MINERAL);
```

Then you declare variables like the following:

```
VAR FLAG : HUES;
    VOWELS : CAPS;
    LIVE : MATTER;
```

Finally, you could assign these set variables:

```
FLAG := [RED, WHITE, BLUE];
VOWELS := ['A', 'E', 'I', 'O', 'U'];
LIVE := [ANIMAL, VEGETABLE];
```

The set elements must be enclosed in brackets. The ORD value of the base type can range from Ø to 255. Thus, SET OF CHAR is legal, but SET OF 1942..1984 is not allowed. If the range checking switch is on, passing a set as a value parameter invokes a runtime compatibility check, unless the formal and actual sets have the same type. Sets provide a clear and efficient way of giving several qualities or attributes to an object. For example:

```
QUALITIES = SET OF (READY, GETSET, ACTIVE, DONE);
```

You could then assign the qualities with X := [GETSET, ACTIVE] and test them with the following operations:

```
IN          tests a bit
+           sets a bit
-           clears a bit
```

For example, an appropriate construction might be:

```
IF ACTIVE IN X THEN WRITELN ('GO FISH')
```

You can also use SET OF Ø..15 to test and set the bits in a WORD. Using WORDs both as a set of bits and as the WORD type requires giving two types to the word, with a variant record, the RETYPE function, or an address type.

Files

A file is a structure that consists of a sequence of components, all of the same type. You must declare a file variable in order to use it. However, the number of components in a file is not fixed by declaring a FILE type.

Examples of FILE declarations:

```
TYPE F1 = FILE OF COLOR;
     F2 = FILE OF CHAR;
     F3 = TEXT;
```

In Pascal, a file is conceptually another data type, like an array, but with no bounds and with only one component accessible at a time. However, a file usually corresponds to one of the following:

1. disk files
2. terminals
3. printers
4. other input and output devices

This implies the following restriction in Pascal: a FILE OF FILE is invalid, directly or indirectly. Other structures, such as a FILE OF ARRAYs or an ARRAY OF FILEs, are permitted.

Pascal supports normal statically allocated files, files as local variables (allocated on the stack), and files as pointer referents (allocated on the heap). Except for files in super arrays, the compiler generates code to initialize a file when it is allocated and to CLOSE a file when it is deallocated.

Except for standard files INPUT and OUTPUT, files in a program header must be given an operating system filename when you run your program. You may use the ASSIGN and READFN procedures to give explicit operating system filenames to files not included in the program header.

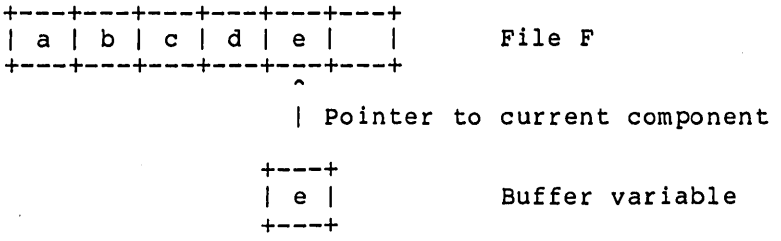
Files in record variants or super array types are not recommended; if you use them, the compiler issues a warning. A file variable cannot be assigned, compared, or passed by value: it can only be declared and passed as a reference parameter.

You can also indicate a file's access method or other characteristics by specifying the mode of the file. The mode is a value of the predeclared enumerated type FILEMODES. The modes include the three base modes, SEQUENTIAL, TERMINAL, and DIRECT. All files, except INPUT and OUTPUT, are given SEQUENTIAL mode by

default. INPUT and OUTPUT are given the default mode TERMINAL.

The Buffer Variable

Every file F has an associated buffer variable F^. The procedures GET and PUT use this buffer variable to READ from and WRITE to files. GET copies the current component of the file to the buffer variable. PUT does the opposite; that is, PUT copies the value of the buffer variable to the current component. A buffer variable and its associated file might look like this:



The buffer variable can be referenced (i.e., its value fetched or stored) like any other variable. This allows execution of assignments like the following:

```
F^ := 'z'
C := F^
```

A file buffer variable can be passed as a reference parameter to a procedure or function or used as a record in a WITH statement. However, the file buffer variable may not be updated correctly if the file position changes within the procedure, function, or WITH statement.

For example, the following use of a file buffer variable would generate a warning at compile time:

```
VAR  A : TEXT;
PROCEDURE CHAR_PROC (VAR X : CHAR);
  .
  .
CHARPROC (A^);      {Warning issued here}
```

File Structures

Files have two basic structures: BINARY and ASCII. These two structures correspond to raw data files and human-readable textfiles, respectively.

BINARY Structure Files

The data type FILE OF type corresponds to BINARY structure files. These, in turn, correspond to unformatted operating system files. Every record is one component of the file type (not to be confused with the Pascal record type). Primitive procedures such as GET and PUT operate on a record basis.

ASCII Structure Files

The data type TEXT corresponds to ASCII structure files. These, in turn, correspond to textual operating system files (called "textfiles"). The Pascal TEXT type is like a FILE OF CHAR, except that groups of characters are organized into "lines" and, to a lesser extent, "pages." Primitive file procedures, such as GET and PUT, always operate on a character basis.

Textfiles (files of type TEXT) are divided into lines with a "line marker," conceptually a character not of the type CHAR. Although a text file can in theory contain any value of type CHAR, writing a particular character (say, CHR (13), carriage return, or CHR (1Ø), line feed) may terminate the current line (record). This character value is the line marker in this case and, when read, always looks like a blank.

A declaration for a text file may include an optional line length. Setting the line length, which sets record length, is only needed for DIRECT mode textfiles. You may specify line length for other modes as well, but doing so has no effect. You must specify the line length of a text file as a constant in parentheses after the word TEXT:

```
TYPE NAMEADDR = TEXT (128);
     DEFAULTX = TEXT;
     SMALLBUF = TEXT (2);
```

File Access Modes

The file modes in are SEQUENTIAL, TERMINAL, and DIRECT. SEQUENTIAL and TERMINAL mode ASCII structure files can have variable length records (lines); DIRECT mode files must have fixed length records or lines.

The declaration of a file in implies its structure, but not its mode. For example, FILE OF STRING (8Ø) indicates BINARY structure; TEXT indicates ASCII structure. An assignment like F.MODE := DIRECT sets the mode; and is needed only to set the DIRECT mode.

TERMINAL Mode Files

TERMINAL mode files always correspond to an interactive terminal or printer. TERMINAL mode files, like SEQUENTIAL mode files, are opened at the beginning of the file for either reading or writing. Records are accessed one after the other until the end of the file is reached.

Operation of TERMINAL mode input for terminals depends on the file structure (ASCII or BINARY). For ASCII structure (type TEXT), entire lines are read at one time. This permits the usual operating system intraline editing, including backspace, advance cursor, and cancel. Characters are echoed to the terminal screen while the line is being typed.

For BINARY structure TERMINAL mode (usually type FILE OF CHAR), you can read characters as you type them. No intraline editing or echoing is done. This method permits screen editing, menu selection, and other interactive programming on a keystroke rather than line basis.

TERMINAL mode files use lazy evaluation to properly handle normal interactive reading of the terminal keyboard. (See "Lazy Evaluation," under chapter 12, File Oriented Procedures and Functions for details.)

SEQUENTIAL Mode Files

SEQUENTIAL mode files are generally disk files or other sequential access devices. Like TERMINAL mode files, SEQUENTIAL mode files are opened at the beginning of the file for either reading or writing, and records are accessed one after another until the end of the file.

DIRECT Mode Files

DIRECT mode files are generally disk files or other random access devices. DIRECT mode ASCII structure files, as well as all BINARY structure files, have fixed-length records, where a record is either a line or file component. (Here the term "record" refers not to the normal Pascal record type, but to a disk structuring unit.) DIRECT files are always opened for both reading and writing, and records can be accessed randomly by record number. There is no record number zero; records begin with record number one.

The Predeclared Files INPUT and OUTPUT

The INPUT and OUTPUT files, are predeclared in every Pascal program. These files get special treatment as program parameters and are normally required as parameters in the program heading:

```
PROGRAM ACTION (INPUT, OUTPUT);
```

If there are no program parameters and the program does not use the files INPUT and OUTPUT, the heading can look like this:

```
PROGRAM ACTION;
```

However, you should include INPUT and OUTPUT as program parameters if you use them, either explicitly or implicitly, in the program itself:

```
WRITE (OUTPUT, 'Prompt: ')  {Explicit use}
WRITE ('Prompt: ')          {Implicit use}
```

These examples would generate a warning if OUTPUT was not declared in the program heading. The only effect of INPUT and OUTPUT as program parameters is to suppress this warning. Although you may redefine the identifiers INPUT and OUTPUT, the file assumed by text file input and output procedures and functions (e.g., READ, EOLN) is the predeclared definition.

The procedures RESET (INPUT) and REWRITE (OUTPUT) are generated automatically, whether or not INPUT and OUTPUT are present as program parameters (you may also use these procedures explicitly). INPUT and OUTPUT have ASCII structure and TERMINAL mode. They are initially connected to your terminal and opened automatically.

Extended I/O Feature

A file variable is really a record, of type FCBFQQ, called a file control block. At the extended level, a few standard fields in this record help you handle file modes and error trapping. Additional fields and the record type FCBFQQ itself can be used at the system level, described under "System I/O Feature." Along with access to certain FCB fields, extended I/O Feature also includes the following procedures:

```
ASSIGN      READFN
CLOSE       READSET
DISCARD     SEEK
```

You should use the normal record field syntax to access FCB fields. For a file F, the fields are named F.MODE, F.TRAP, and F.ERRS. You may change or examine these fields at any time.

1. F.MODE: FILEMODES

This field contains the mode of the file: SEQUENTIAL, TERMINAL, or DIRECT. These values are constants of the predeclared enumerated type FILEMODES. The file system uses the MODE field only during RESET and REWRITE. Thus, changing the MODE field of an open file has no effect and is, in fact, discouraged. Except for INPUT and OUTPUT, which have TERMINAL mode, a file's mode is SEQUENTIAL by default.

2. F.TRAP: BOOLEAN

If this field is TRUE, error trapping for file F is turned on. Then, if an input/output error occurs, the program does not abort and the error code can be examined. Initially, F.TRAP is set FALSE. If FALSE and an I/O error occurs, the program aborts. Closing the file sets the trap to false. Note that reset and rewrite close the file

3. F.ERRS: WRD(Ø)..15

This field contains the error code for file F. An error code of zero means no error; values from 1 to 15 imply an error condition. If you attempt a file operation other than CLOSE or DISCARD and F.ERRS is not zero, the program immediately aborts if F.TRAP is FALSE. However, if F.TRAP is TRUE, the attempted file operation is ignored and the program continues.

CLOSE and DISCARD do not examine the initial value of F.ERRS, so they are never ignored and do not cause an immediate abort. If CLOSE or DISCARD themselves generate an error condition, F.TRAP is used to determine whether to trap the error or to abort.

An operation ignored because of an error condition does not change the file itself, but may change the buffer variable or READ procedure input variables. See Appendix D, "Error Messages," for a complete listing of error messages and warnings.

The Extended I/O Feature, allows to you set the line length for a text file, as follows:

```
TYPE SMALLBUF = TEXT (16);
VAR RANDOMTEXT: TEXT (132);
```

Declaring line length applies only to DIRECT mode ASCII structure files, where the line length is the record length used for reading and writing. Setting the line length has no effect on other ASCII files.

System Level I/O

The System I/O Feature allows you to call procedures and functions that have a formal reference parameter of type FCBFQQ with an actual parameter of the type FILE OF type or TEXT, or the identical FCBFQQ type.

The FCBFQQ type is the underlying record type used to implement the file type. The interface for the target file system FCBFQQ type (and any other types needed) is usually part of the internal file system. Thus, procedures and functions that reference FCBFQQ parameters can be called with any file type, including predeclared procedures and functions like CLOSE and READ.

Reference Types

A reference to a variable or constant is an indirect way to access it. The pointer type is an abstract type for creating, using, and destroying variables allocated from an area called the heap. The heap is a dynamically growing and shrinking region of memory allocated for pointer variables.

Pascal also provides two machine-oriented address types: one for addresses that can be represented in 16 bits, the other for addresses that require 32 bits.

Pointers are generally used for trees, graphs, and list processing. Use of pointers is portable, structured, and relatively safe.

Address types provide an interface to the hardware and operating system; their use is frequently unstructured, low level, and unsafe.

Pointer Types

A pointer type is a set of values that point to variables of a given type. The type of the variables pointed to is called the "reference type." Reference variables are all dynamically allocated from the heap with the NEW procedure. Pascal variables are normally allocated on the stack or at fixed locations.

You can perform only the following actions on pointers:

1. assign them
2. test them for equality and inequality with the two operators = and <>
3. pass them as value or reference parameters

4. dereference them with the up arrow (^)

Every pointer type includes the pointer value NIL. Pointers are frequently used to create list structures of records, as shown in the following example:

```
TYPE
  TREETIP = ^ TREE;
  TREE = RECORD
           VAL: INTEGER;
              {Value of TREE cell.}
           LEFT, RIGHT:  TREETIP
              {Pointers to other TREETIP cells.}
              {Note recursive definition.}
         END;
```

Unlike most type declarations, a pointer type can refer to a type of which it is itself a component. The declaration can also refer to a type declared later in the same TYPE section, as in TREE and TREETIP in the previous example.

Such a declaration is called a forward pointer declaration and permits recursive and mutually recursive structures. Because pointers are so often used in list structures, forward pointer declarations occur frequently.

The compiler checks for one ambiguous pointer declaration. Suppose the previous example was in a procedure nested in another procedure that also declared a type TREE. Then the reference type of TREETIP could be either the outer definition or the one following in the same TYPE section. The compiler assumes the TREE type intended is the one later in the same TYPE section and gives the warning:

Pointer Type Assumed Forward

A pointer can have a super array type as a referent type. The actual upper bounds of the array are passed to the NEW procedure to create a heap variable of the correct size. Forward pointer declarations of the super array type are not allowed.

You cannot declare two pointers with different types and then assign or compare them, even if they happen to point to the same underlying type. For example:

```
VAR PRA : ^ REAL;
    PRE : ^ REAL;
BEGIN PRA := PRE END; {This is illegal!}
```

Programs usually contain only one type declaration for a pointer to a given type. In the TREETIP example, the type of LEFT and RIGHT could be ^TREE instead of TREETIP, but then you couldn't

assign variables of type TREETIP to these fields. However, it is sometimes useful to make sure that two classes of pointers are not used together, even if they point to the same type.

For example, suppose you have a type RESOURCE kept in a list and declare two types, OWNER and USER, of type ^RESOURCE. The compiler would catch assignment of OWNER values to USER variables and vice versa and issue a warning message.

If the $INITCK is on, a newly created pointer has an uninitialized value. If the NIL checking switch is on, pointer values are tested for various invalid values. Invalid values include NIL, uninitialized values, reference to a heap item that has been DISPOSEd, or a value that is not valid as a heap reference.

Address Types

The keywords ADR and ADS refer to the relative address type and the segmented address type respectively. As the following example shows you can use the keywords both as type clause prefixes and as prefix operators:

```
VAR INT_VAR  :  INTEGER;
    REAL_VAR :  REAL;
    A_INT    :  ADR OF INTEGER;{Declaration of ADR variable}

    AS_REAL  :  ADS OF REAL;   {Declaration of ADS variable}

BEGIN
  INT_VAR  :=  1;            {Integer variable}
  REAL_VAR :=  3.1415;       {Real variable}
  A_INT    :=  ADR INT_VAR; {ADR used as operator}
  AS_REAL  :=  ADS REAL_VAR;{ADS used as operator}
  WRITELN (A_INT^,AS_REAL^) {Up arrow used to dereference
                             the address types.}
END.
```

You may declare a variable that is an address:

```
VAR X :  ADR OF BYTE;
```

Then, with the following record notation, you can assign numeric values to the actual variable:

```
X.R := 16#FFFF
```

You may specify the assigned value in hexadecimal notation. You may also assign to a segment field with the ADS type, using the field notation .S (segment address). Thus, you may declare a variable of an ADS type and then assign values to its two fields:

```
VAR Y : ADS OF WORD;
  .
  .
Y.S := 16#0001
Y.R := 16#FFFF
```

As shown above, any 16-bit value can be directly assigned to address type variables, using the .R and .S fields. The ADR and ADS operators obtain these addresses directly. The example below assigns addresses this way to the variables X and Y:

```
VAR  X : ADR OF BYTE;
     Y : ADS OF WORD;
     W : WORD;
     B : BYTE;
       .
       .
     X := ADR B;
     Y := ADS W;
```

Pascal supports the following predeclared address types:

```
ADRMEM = ADR OF ARRAY [0..32766] OF BYTE;
ADSMEM = ADS OF ARRAY [0..32766] OF BYTE;
```

Since the type referred to by the address is an array of bytes, byte indexing is possible. For example, if A is of type ADRMEM, then A^[15] is the byte at the address A.R + 15, where .R specifies an actual 16-bit address.

You can use the address types for a constant address (a form of structured constant); you may also take the address of a constant or expression. For example:

```
TYPE ADRWORD = ADR OF WORD;
     ADSWORD = ADS OF WORD;
VAR W: WORD;
    R: ADRWORD;
CONST CONADR = ADRWORD (1234);
BEGIN
  W := CONADR^;              {Get word at address 1234}
  W := ADSWORD (0, 32)^;     {Get word at address 0:32}
  W := (ADS W).S;            {Get value of DS segment register}
  R := ADR '123';            {Get address of a constant value}
  R := ADR (W DIV 2 + 1); {Get address of expression value}
END;
```

However, constants or expressions that yield addresses cannot be used as the target of an assignment (or as a reference parameter or WITH record), as shown:

```
CONST ADSCON = ADSWORD (256, 64);    {Valid}
FUNCTION SOME_ADDRESS: ADSWORD;      {Valid}
BEGIN
  ADSWORD (Ø, 32)^ := W; {Invalid}
  ADSCON^ := 12;         {Invalid}
  SOME_ADDRESS^ := 1ØØ;  {Invalid}
END;
```

Segment Parameters for the Address Types

Two keywords, VARS and CONSTS, are available as parameter prefixes, like VAR and CONST, to pass the segmented address of a variable. If P is of type ADS FOO, then P^ can be passed to a VARS formal parameter, such as VARS X: FOO, but cannot be passed to a VAR formal parameter.

In the B 2Ø environment, a default data segment is assumed, in which case a VAR parameter is passed as the default data segment offset of a variable. A VARS parameter is passed as both the segment value and the offset value. Both VARS parameters and ADS variables have the offset (.R) value in the WORD with the lower address and the segment (.S) value in the address plus two.

In pointer type declarations, the up arrow (^) prefixes the type pointed to; in program statements, it dereferences a pointer so that the value pointed to can be assigned or operated on. The up arrow also dereferences ADR and ADS types in program statements.

Component selection with the up arrow (^) is performed before the unary operators ADR or ADS. Because the up arrow (^) selector can appear after any address variable to produce a new variable, it can occur, for example, in the target of an assignment, a reference parameter, as well as in expressions. Since ADS and ADR are prefix operators, they are used only in expressions, where they apply only to a variable or constant or expression.

Using the Address Types

The following example illustrates the rules that you must follow to combine and intermingle the two address types:

```
VAR
   P: ADS OF DATA;  {P is segmented address of type DATA.}
   Q: ADR OF DATA;  {Q is relative address of type DATA.}
   X: DATA;         {X is some variable of type DATA.}

BEGIN
  P := ADS X;       {Assign the address of X to P.}

  X := P^;          {Assign to X the value pointed to by P.}
```

```
    P := ADS P^;          {Assign to P the address of the value
                           whose address is pointed to by P.
                           P is unchanged by this assignment.}

    Q := ADR X;           {Assign the relative address of X to Q.}

    Q.R := (ADR X).R;     {Assign the relative address of X to Q,
                           using the WORD type.}

    P := ADS Q^;          {Assign address of variable at Q to P.}

    Q := ADR P^;          {Invalid; you cannot apply ADR to ADS
                           ^.}

    P.R := 16#8000;       {Assign 32768 to P's offset field.}

    P.S := 16;            {Assign 16 to P's segment field.}

    Q.R := P.R + 4;       {Assign P's offset plus 4 to be the
                           value of Q.}
END;
```

The address type and pointer type should be treated as two distinct types. The pointer type, in theory, is just an undefined mapping from a variable to another variable. The method of implementation is undefined. However, the address type deals with actual machine addresses.

The following special facilities that use pointer variables are not allowed with address variables.

1. The NEW and DISPOSE procedures are only permitted with pointers. NIL does not apply to the address type. There are no special address values for empty, uninitialized, or invalid addresses.

2. The type "address of super array type" is not supported in the same way as "pointer to super array type." Getting the address of a super array variable is still permitted with ADR and ADS. For example, if a procedure or function formal parameter is declared as VAR S: STRING, then within the procedure or function, the expression ADS S is fine. Unlike a pointer, the address does not contain any upper bounds.

Packed Types

Any of the structured types can be PACKED. This could economize storage at the possible expense of access time or access code

space. However, the following limitations apply on the use of PACKED structures:

1. The prefix PACKED is always ignored, except for type checking, in sets, files, and arrays of characters, and has no actual effect on the representation of records and other arrays. Furthermore, PACKED can only precede one of the structure names ARRAY, RECORD, SET, or FILE; it cannot precede a type identifier. For example, if COLORMAP is the identifier for an unpacked array type, "PACKED COLORMAP" is not accepted.

2. A component of a PACKED structure cannot be passed as a reference parameter or used as the record of a WITH statement, unless the structure is of a string type. Also, obtaining the address of a PACKED component with ADR or ADS is not permitted.

3. A PACKED prefix only applies to the structure being defined: any components of that structure that are also structures are not packed unless you explicitly include the reserved word PACKED in their definition.

Note that the operators ADS and ADR do not apply to procedures. However, the address of a procedure can be computed as follows. Suppose a Pascal program contains a public procedure Proc declared, for example as:

```
PROCEDURE Proc (w: WORD) [PUBLIC];
```

To compute the ADS of this procedure, declare an external function GetProc, whose only parameter is a procedure with the same arguments as Proc. For example:

```
TYPE
     pProcType = ADS of WORD;

FUNCTION GetProc (PROCEDURE Proc(w: WORD)): pProcType;
                    Extern;
```

Then link into the program a Pascal module containing:

```
TYPE
     pProcType = ADS of WORD;
     opProcType = ADR of pProcType;

FUNCTION GetProc (opProc: opProcType; wJunk: WORD):
                    pProcType;

  BEGIN
    GetProc := opProc?;
  END;
```

Procedural and Functional Types

Procedural and functional types are different from other types. (Wherever the term "procedural" is used from here on, both procedural and functional is implied.) You may not declare an identifier for a procedural type in a TYPE section; nor may you declare a variable of a procedural type. However, you may use procedural types to declare the type of a procedural parameter, and in this sense they conform to the Pascal idea of a type.

A procedural type defines a procedure or function heading and gives any parameters. For a function, it also defines the result type. The syntax of a procedural type is the same as a procedure or function heading, including any attributes.

Example of a procedural type declaration:

```
PROCEDURE ZERO (FUNCTION FUN (X, Y: REAL): REAL)
```

Type Compatibility

The type compatibility is the same as ISO Pascal with some additional rules added for super array types, LSTRINGs, and constant coercions (i.e., forced changes in the type of a constant). Type transfer functions, to override the typing rules, are available in some cases like ORD and RETYPE.

Two types can be "identical," "compatible," or "incompatible." An expression may or may not be "assignment compatible" with a variable, value parameter, or array index.

Type Identity and Reference Parameters

Two types are identical if they have the identical identifier or if the identifiers are declared equivalent with a type definition like the following:

```
TYPE T1 = T2;
```

There is no difference between types T1 and T2 in the example above. Type identity is based on the name of the types, and not on the way they are declared or structured. Thus, for example, T1 and T2 are not identical in the following declarations:

```
TYPE T1 = ARRAY [1..10] OF CHAR;
     T2 = ARRAY [1..10] OF CHAR;
```

Actual and formal reference parameters must be of identical types. Or, if a formal reference parameter is of a super array type, the actual parameter must be of the same super array type or a type derived from it. Two record or array types must be identical for assignment.

The only exception is for strings. Here, actual parameters of type CHAR, STRING, STRING (n), LSTRING, and LSTRING (n) are compatible with a formal parameter of super array type STRING. Also, the type of a string constant will change to any LSTRING type with a large enough bound. For example, the type of 'ABC' will change to LSTRING (5) if necessary.

Furthermore, an actual parameter of any FILE type may be passed to a formal parameter of of a special record type FCBFQQ. Similarly, an actual parameter of type FCBFQQ may be passed to a formal parameter of any file type.

STRING (n) is a shorthand notation for:

```
PACKED ARRAY [1..n] OF CHAR
```

The two types are identical. However, because variables with the type LSTRING are treated specially in assignments, comparisons, READs, and WRITEs, LSTRING (n) is not a shorthand notation for PACKED ARRAY [Ø..n] OF CHAR. The two types are not identical, compatible, or assignment compatible.

Type Compatibility and Expressions

Two simple or reference types are compatible if:

1. They are identical.
2. They are both ADR types.
3. They are both ADS types.
4. One is a subrange of the other.
5. They are subranges of compatible types.

Two structured types are compatible if:

1. They are identical.
2. They are SET types with compatible base types.
3. They are STRING derived types of equal length.
4. They are LSTRING derived types.

However, two structured types are incompatible if:

1. Either type is a FILE or contains a FILE.

2. Either type is a super array type.

3. One type is PACKED and the other is not.

Two values must be of compatible types when combined with an operator in an expression. (Most operators have additional limitations on the type of their operands. See Chapter 8, "Expressions," for details.) A CASE index expression type must be compatible with all CASE constant values. Note that two sets are never compatible if one is PACKED and the other is not.

Assignment Compatibility

Some types are implicitly compatible. This permits assignment across type boundaries. For instance, assume you declare the following variables:

```
VAR DESTINATION : T_DEST;
    SOURCE      : T_SOURCE;
```

SOURCE is assignment compatible with DESTINATION (i.e., DESTINATION := SOURCE is permitted) if one of the following is true:

1. T_SOURCE and T_DEST are identical types.

2. T_SOURCE and T_DEST are compatible and SOURCE has a value in the range of subrange type T_DEST.

3. T_DEST is of type REAL and T_SOURCE is compatible with type INTEGER or INTEGER4.

4. T_DEST is of type INTEGER4 and T_SOURCE is compatible with type INTEGER or WORD.

Also, if T_DEST and T_SOURCE are compatible structured types, then SOURCE is assignment compatible with DESTINATION if one of the following is true:

1. For SETs, every member of SOURCE is in the base type of T_DEST.

2. For LSTRINGs, UPPER (DESTINATION) >= SOURCE.LEN.

Other than in the assignment statement itself, assignment compatibility is required in the following cases of implicit assignment:

1. passing value parameter

2. READ and READLN procedures

3. control variable and limits in a FOR statement

4. super array type array bounds, and array indices

Assignment compatibility is usually known at compile time, and an assignment generates simple instructions. However, some subrange, set, and LSTRING assignments depend on the value of the expression to be assigned and thus cannot be checked until runtime. If the $RANGECK is on, assignment compatibility is checked at runtime; otherwise, no checking is done.

CHAPTER 7
VARIABLES AND VALUES

CONTENTS

A variable is a value that is expected to change during the course of a program. Every variable must be of a specific data type. A variable may have an identifier. If A is a variable of type INTEGER, then the use of A in a program actually refers to the data denoted by A.

For example:

```
VAR A: INTEGER;
   BEGIN
      A := 1;
      A := A + 1;
   END;
```

These statements would first assign a value of 1 to the data denoted by A, and subsequently assign it a value of 2.

Variables are manipulated by using some sort of notation to denote the variable, such as a variable identifier. In other cases, variables may be denoted by array indices or record fields or the dereferencing of pointer or address variables. The compiler itself may sometimes create "hidden" variables, allocated on the stack, in circumstances like the following:

1. When you call a function that will return a structured result, the compiler allocates a variable in the caller for the result.

2. When you need the address of an expression (e.g., to pass it as a reference parameter or to use it as a WITH statement record or with ADR or ADS), the compiler allocates a variable for the value of the expression.

3. The initial and final values of a FOR loop may require allocating a variable.

4. When the compiler evaluates an expression, it may allocate a variable to store intermediate results.

5. Every WITH statement requires a variable to be allocated for the address of the WITH's record.

VARIABLE DECLARATIONS

A variable declaration consists of the identifier for the new variable, followed by a colon and a type. You may declare variables of the same type by giving a list of the variable identifiers, followed by their common type. For example:

```
VAR XCOORD, YCOORD:  REAL
```

You may declare a variable in any of the following locations:

1. VAR section of a program, procedure, function, module, interface, or implementation

2. formal parameter list of a procedure, function, or procedural parameter

In a VAR section, you may declare a variable to be of any valid type; in a formal parameter list, you may include only a type identifier (i.e., you cannot declare a type in the heading of a procedure or function). For example:

```
PROCEDURE NAME (GEORGE:  ARRAY [1..10] OF COLOR)
                         {Invalid; GEORGE is of a new type.}

VAR VECTOR_A:  VECTOR (10)
                         {Valid; VECTOR (10) is a type
                         derived from a super type.}
```

Each declaration of a file variable F of type FILE OF T implies the declaration of a buffer variable of type T, denoted by F^. A file declaration also implies the declaration of a record variable of type FCBFQQ, whose fields are denoted as F.TRAP, F.ERRS, F.MODE, and so on. See chapter 12, File Oriented Procedures and Functions for further information on buffer variables and FCBFQQ fields.

THE VALUE SECTION

The VALUE section allows you to give initial values to variables in a program, module, procedure, or function. You may also initialize the variable in an implementation, but not in an interface.

The VALUE section may include only statically allocated variables, that is, any variable declared at the program, module, or implementation level, or a variable with the STATIC or PUBLIC attribute. Variables with the EXTERN or ORIGIN attribute cannot occur in a VALUE section, since they are not allocated by the compiler.

The VALUE section may contain assignments of constants to entire variables or to components of variables. For example:

```
VAR  ALPHA : REAL;
     ID    : STRING (7);
```

```
    I     : INTEGER;

VALUE
    ALPHA := 2.23;
    ID[1] := 'J';
    I     := 1;
```

USING VARIABLES AND VALUES

A denotation of a variable may designate one of three things:

1. an entire variable
2. a component of a variable
3. a variable referenced by a pointer

A value may be any of the following:

1. a variable
2. a constant
3. a function designator
4. a component of a value
5. a variable referenced by a reference value

A function can also return an array, record, or set. The same syntax used for variables may be used to denote components of the structures these functions return.

This feature also allows you to dereference a reference type that is returned by a function. However, you can only use the function designator as a value, not as a variable. For example, the following is invalid:

```
F (X, Y)^ := 42;
```

You may declare constants of a structured type. Components of a structured constant use the same syntax as variables of the same type.

Examples of structured constant components:

```
TYPE REAL3 = ARRAY [1..3] OF REAL;      {an array type}
CONST PIES = REAL3 (3.14, 6.28, 9.42); {an array constant}
  .
  .
  X := PIES [1] * PIES [3];             {i.e., 3.14 * 9.42}
```

```
Y := REAL3 (1.1, 2.2, 3.3) [2];        {i.e., 2.2}
```

Components of Entire Variables and Values

A variable identifier denotes an entire variable. A variable, function designator, or constant denotes an entire value. A component of a variable or value is denoted by the identifier followed by a selector that specifies the component. The form of a selector depends on the type of structure (array, record, file, or reference).

Indexed Variables and Values

A component of an array is denoted by the array variable or value, followed by an index expression. The index expression must be assignment compatible with the index type in the array type declaration. An index type must always be an ordinal type. The index itself must be enclosed in brackets following the array identifier.

Examples of indexed variables and values:

```
ARRAY_OF_CHAR ['C']        {Denotes the Cth element.}

'STRING CONSTANT' [6] := 'G'
                           {Assigns the 6th element, the letter
                           'G'.}

ARRAY_FUNCTION (A, B) [C, D]
                           {Denotes a component of a two-
                           dimensional array returned by
                           ARRAY_FUNCTION (A, B). A and B are
                           actual parameters.}
```

You may specify the current length of an LSTRING variable, LSTR, in either of two ways:

1. with the notation LSTR [Ø], to access the length as a CHAR component

2. with the notation LSTR.LEN, to access the length as a BYTE value

Field Variables and Values

A component of a record is denoted by the record variable or value followed by the field identifier for the component. Fields are separated by the period (.). In a WITH statement, you give the record variable or value only once. Within the WITH statement, you may use the field identifier of a record variable

directly.

Examples of field variables and values:

```
PERSON.NAME := 'PETE'

PEOPLE.DRIVERS.NAME := 'JOAN'

WITH PEOPLE.DRIVERS DO
  NAME := 'GERI'

RECURSING_FUNC ('XYZ').BETA
                          {Selects  BETA  field  of  record
                          returned  by  the  function  named
                          RECURSIVE_FUNC.}

COMPLEX_TYPE (1.2, 3.14).REAL_PART
```

Record field notation also applies to files for FCBFQQ fields, to address type values for numeric representations, and to LSTRINGs for the current length.

File Buffers and Fields

At any time, only one component of a file is accessible. The accessible component is determined by the current file position and represented by the buffer variable. Depending on the status of the buffer variable, fetching its value may first read the value from the file. (This is called "lazy evaluation"; see Chapter12, File oriented procedures and Functions for more information).

If a file buffer variable is passed as a reference parameter or used as a record of a WITH statement, the compiler issues a warning to alert you to the fact that the value of the buffer variable may not be correct after the position of the file is changed with a GET or PUT procedure.

Examples of file reference variables:

```
INPUT^
ACCOUNTS_PAYABLE.FILE^
```

Reference Variables

Reference variables or values denote data that refers to some data type. There are three kinds of reference variables and values:

1. pointer variables and values

2. ADR variables and values

3. ADS variables and values

In general, a reference variable or value "points" to a data object. Thus, the value of a reference variable or value is a reference to that data object. To obtain the actual data object pointed to, you must "dereference" the reference variable by appending an up arrow (^) to the variable or value.

Example using pointer values:

```
VAR   P, Q : ^INTEGER;  {P and Q are pointers to integers.}

NEW (P); NEW (Q);       {P and Q are assigned reference
                         values to regions in memory
                         corresponding to data objects of
                         type INTEGER.}

P := Q;                 {P and Q now point to the same
                         region in memory.}

P^ := 123;              {Assigns the value 123 to the
                         INTEGER value pointed to by P.
                         Since Q points to this location as
                         well, Q^ is also assigned 123.}
```

Using NIL^ is an error (since a NIL pointer does not reference anything). You may also append an up arrow (^) to a function designator for a function that returns a pointer or address type. In this case, the up arrow denotes the value referenced by the return value. This variable cannot be assigned to or passed as a reference parameter.

Examples of functions returning reference values:

```
DATA1 := FUNK1 (I, J)^  {FUNK1 returns a reference value.
                         The up arrow dereferences the
                         reference value returned, assigning
                         the referenced data to DATA1.}

DATA2 := FUNK2 (K, L)^.FOO [2]
                        {FUNK2 returns a reference value.
                         The up arrow dereferences the
                         reference value returned.  In this
                         case, the dereferenced value is a
                         record.  The array component FOO [2]
                         of that record is assigned to the
                         variable DATA2.}
```

If P is of type ADR OF some type, then P.R denotes the address value of type WORD. If P is of type ADS OF some type, then P.R denotes the offset portion of the address and P.S denotes the segment portion of the address. Both portions are of type WORD.

Examples of address variables:

```
BUFF_ADR.R
DATA_AREA.S
```

ATTRIBUTES

A variable declaration or the heading of a procedure or function may include one or more attributes. A variable attribute gives special information about the variable to the compiler.

The following attributes are provided for variables:

Attribute	Variable
STATIC	Allocated at a fixed location, not on the stack.
PUBLIC	Accessible by other modules with EXTERN, implies STATIC.
EXTERN	Declared PUBLIC in another module, implies STATIC.
ORIGIN	Located at specified address, implies STATIC.
READONLY	Cannot be altered or written to.

The EXTERN attribute is also a procedure and function directive; PUBLIC and ORIGIN are also procedure and function attributes. See Chapter 10, Procedures and Functions for a discussion of procedure and function attributes and directives.

You may only give attributes for variables in a VAR section. Specifying variable attributes in a TYPE section or a procedure or function parameter list is not permitted.

You can give one or more attributes in the variable declaration, enclosed in brackets and separated by commas (if specifying more than one attribute).

The brackets may occur in either of two places:

1. An attribute in brackets after a variable identifier in VAR section applies to that variable only.

2. An attribute in brackets after the reserved word VAR applies to all of the variables in the section.

Examples that specify variable attributes:

```
VAR A, B, C [EXTERN] : INTEGER;  {Applies to C only.}

VAR [PUBLIC] A, B, C  : INTEGER; {Applies to A, B, and C.}

VAR [PUBLIC] A, B, C [ORIGIN 16#1000] : INTEGER;
                        {A, B, and C are all PUBLIC.  ORIGIN
                        of C is the absolute hexadecimal
                        address 1000.
```

The STATIC Attribute

The STATIC attribute gives a variable a unique, fixed location in memory. This is in contrast to a procedure or function variable that is allocated on the stack or one that is dynamically allocated on the heap. Use of STATIC variables can save time and code space, but increases data space.

All variables at the program, module, or unit level are automatically assigned a fixed memory location and given the STATIC attribute. Functions and procedures that use STATIC variables can execute recursively, but STATIC variables must be used only for data common to all invocations.

Files declared in a procedure or function with the STATIC attribute are initialized when the routine is entered; they are closed when the routine terminates like other files. However, other STATIC variables are only initialized before program execution. This means that, except for open FILE variables, STATIC variables can be used to retain values between invocations of a procedure or function.

Examples of STATIC variable declarations:

```
VAR VECTOR [STATIC]: ARRAY [0..MAXVEC] OF INTEGER;
VAR [STATIC] I, J, K: 0..MAXVEC;
```

The STATIC attribute does not apply to procedures or functions, as some other attributes do.

The PUBLIC and EXTERN Attributes

The PUBLIC attribute indicates a variable that may be accessed by other loaded modules; the EXTERN attribute identifies a variable that resides in some other loaded module. Variables given the PUBLIC or EXTERN attribute are implicitly STATIC.

Examples of PUBLIC and EXTERN variable declarations:

```
VAR [EXTERN] GLOBE1, GLOBE2: INTEGER;
                        {EXTERN, meaning that they must be
                        declared PUBLIC in some other loaded
                        module.}

VAR BASE_PAGE [PUBLIC, ORIGIN #12FE]: BYTE;
                        {The variable BASE_PAGE is located
                        at 12FE, hexadecimal. Because it is
                        also PUBLIC, it can be accessed from
                        other loaded modules that declare
                        BASE_PAGE with the EXTERN
                        attribute.}
```

PUBLIC variables are usually allocated by the compiler, unless you also give them an ORIGIN. Giving a variable both the PUBLIC and ORIGIN attributes tells the loader that a global name has an absolute address.

If both PUBLIC and ORIGIN are present, the compiler does not need the loader to resolve the address. However, the identifier is still passed to the linker for use by other modules.

EXTERN variables are not allocated by the compiler. Nor do they have an ORIGIN, since giving both EXTERN and ORIGIN implies two different ways to access the variable. The reserved word EXTERNAL is synonymous with EXTERN.

Variables in the interface of a unit are automatically given either the PUBLIC or EXTERN attribute. If a program, module, or unit USES an interface, its variables are made EXTERN; if you compile the IMPLEMENTATION of the interface, its variables are made PUBLIC.

The ORIGIN Attribute

The ORIGIN attribute directs the compiler to locate a variable at a given memory address. The address must be a constant of any ordinal type. I/O ports, interrupt vectors, operating system data, and other related data can be accessed with the ORIGIN variable.

Example of ORIGIN and STATIC variable declarations:

```
VAR INTRVECT [ORIGIN 8#200]:  WORD;
```

Variables with ORIGIN attribute are implicitly STATIC. Also,

they inhibit common subexpression optimization. For example, if GATE has the ORIGIN attribute, the two statements X := GATE; Y := GATE access GATE twice in the order given, instead of using the first value for both assignments. This ensures correct operation if GATE is a memory-mapped input port. However, if GATE is passed as a reference parameter, references to the parameter may be optimized away.

ORIGIN variables are never allocated or initialized by the compiler. The associated address only indicates where the variable is found. ORIGIN always implies a memory address.

Giving the ORIGIN attribute in brackets immediately following the VAR keyword is ambiguous and generates an error during compilation. (It would be unclear to the compiler whether all variables following should be at the same address or whether addresses should be assigned sequentially.)

```
VAR [ORIGIN Ø] FIRST, SECOND: BYTE; {Invalid}
```

ORIGIN permits a segmented address using "segment : offset" notation.

```
VAR SEGVECT [ORIGIN 16#ØØØ1:16#FFFE]: WORD;
```

A variable with a segmented ORIGIN cannot be used as the control variable in a FOR statement.

The READONLY Attribute

The READONLY attribute prevents assignments to a variable. It also prevents the variable being passed as a VAR or VARS parameter. Also, a READONLY variable cannot be read with a READ statement or used as a FOR control variable. You may use READONLY with any of the other attributes.

Example of READONLY variable declaration:

```
VAR [READONLY] I, J [PUBLIC], K [EXTERN]: INTEGER;
                        {I, J, and K are all READONLY; J is
                        also PUBLIC; K is also EXTERN.}
```

CONST and CONSTS parameters, as well as FOR loop control variables (while in the body of the loop), are automatically given the READONLY attribute. READONLY is the only variable attribute that does not imply STATIC allocation.

A variable that is both READONLY and either PUBLIC or EXTERN in one source file is not necessarily READONLY when used in another source file. The READONLY attribute does not apply to procedures and functions.

Combining Attributes

You may give a variable multiple attributes. Separate the attributes with commas and enclose the list in brackets, as shown:

```
VAR [STATIC]
X, Y, Z [ORIGIN #FFFE, READONLY]: INTEGER;
```

In the above example, Z is a STATIC, READONLY variable with an ORIGIN at hexadecimal FFFE. The following rules apply when you combine attributes:

1. If you give a variable the EXTERN attribute, you should not give it the ORIGIN, or PUBLIC attributes in the current compiland.

2. If you give a variable the ORIGIN attribute, you should not give it the EXTERN attribute. However, you may combine ORIGIN with PUBLIC.

3. If you give a variable the PUBLIC attribute, you should not give it the EXTERN attribute. However, you may combine PUBLIC with ORIGIN.

4. You can use STATIC and READONLY with any other attributes.

CHAPTER 8
EXPRESSIONS

CONTENTS

Expressions are constructions that evaluate to values. For example, the following are all expressions:

```
A + 2

(A + 2)

(A + 2) * (B - 3)
```

The operands in an expression may be a value or any other expression. When any operator is applied to an expression, that expression is called an operand. With parentheses for grouping and operators that use other expressions, you can construct expressions as long and complicated as desired.

Operations follow the rules of operator precedence. There are four precedence laws in the following order:

1. Unary

   NOT [ADR ADS]

2. Multiplying

   * / DIV MOD AND

3. Adding

   + - OR (XOR)

4. Relational

   = <> <= >= < > IN

An expression is either a value or the result of applying an operator to one or two values. Although a value can be of almost any type, most operators only apply to the following types:

INTEGER	INTEGER4
WORD	BOOLEAN
REAL	SET

The relational operators also apply for the CHAR, enumerated, string, and reference types. For all operators (except the set operator IN), operands must have compatible types.

SIMPLE EXPRESSIONS

As a rule, the operands and the value resulting from an operation are all of the same type. Occasionally, however, the type of an operand is changed to the type required by an operator.

This conversion occurs on two levels: one for constant operands only, and one for all operands. INTEGER to WORD conversion occurs for constant operands only; conversion from INTEGER to REAL and from INTEGER or WORD to INTEGER4 occurs for all operands.

If necessary in constant expressions, INTEGER values change to WORD type. You should be cautious when mixing INTEGER and WORD constants in expressions. For example, if CBASE is the constant 16#CØØØ and DELTA is the constant -1, the following expression gives a WORD overflow:

```
WRD (CBASE) + DELTA
```

The overflow occurs because DELTA is converted to the WORD value 16#FFFF, and 16#CØØØ plus 16#FFFF is greater than MAXWORD. However, the following would work:

```
WRD (ORD (CBASE) + DELTA)
```

This expression gives the INTEGER value -16385, which changes to WORD 16#BFFF. If conversion is needed by an operator or for an assignment, the compiler makes the following conversions:

1. from INTEGER to REAL or INTEGER4
2. from WORD to INTEGER4

The following rules determine the type of the result of an expression involving these simple types:

1. + - *

   These operators operate on INTEGERs, REALs, WORDs, and INTEGER4s, as shown in the following examples:

   ```
   +123
   A + 123
   -23.4
   A - 8
   A * B * 3
   ```

   Mixtures of REALs with INTEGERs and of INTEGER4s with INTEGERs or WORDs are allowed. Where both operands are of the same type, the result type is the type of the operands. If either operand is REAL, the result type is REAL; otherwise, if either operand is INTEGER4, the result type is INTEGER4.

   Unary plus (+) and minus (-) are supported, along with the binary forms. Unary minus on a WORD type is 2's complement (NOT is 1's complement); since there are no negative WORD values, this always generates a warning.

Because unary minus has the same precedence level as the adding operators:

```
(X * -1)       {Invalid}

(-256 AND X)   {Is interpreted as -(256 AND X)}
```

2. /

This is a "true" division operator. The result is always REAL. Operands may be INTEGER or REAL (not WORD or INTEGER4).

Examples of division:

```
34 / 26.4  =  1.28787...
18 / 6     =  3.00000...
```

3. DIV MOD

These are the operators for integer divide quotient and remainder, respectively. The left operand (dividend) is divided by the right operand (divisor).

Examples of integer division:

```
 123 MOD  5 =  3
-123 MOD  5 = -3  {Sign of result is}
                  {sign of dividend }
 123 MOD -5 =  3
 1.3 MOD  5       {Invalid with REAL operands}
 123 DIV  5 =  24
 1.3 DIV  5       {Invalid with REAL operands}
```

Both operands must be of the same type: INTEGER, WORD, or INTEGER4 (not REAL). The sign of the remainder (MOD) is always the sign of the dividend.

The semantics for DIV and MOD with negative operands, are different from ISO Pascal but the resulting code is more efficient. Programs intended to be portable should not use DIV and MOD unless both operands are positive.

4. AND OR XOR NOT

These operators are bitwise logical functions. Operands must be INTEGER or WORD or INTEGER4 (never a mixture), and cannot be REAL. The result has the type of the operands.

NOT is a bitwise one's complement operation on the

single operand. If an INTEGER variable V has the value MAXINT, NOT V gives the invalid INTEGER value -32768. This generates an error if the $INITCK is on and the value is used later in a program.

Given the following initial INTEGER values,

```
X = 2#1111000011110000
Y = 2#1111111100000000
```

AND, OR, XOR, and NOT perform the following functions:

```
X AND Y     1111000011110000
            1111111100000000
            ----------------
            1111000000000000

X OR Y      1111000011110000
            1111111100000000
            ----------------
            1111111111110000

X XOR Y     1111000011110000
            1111111100000000
            ----------------
            0000111111110000

NOT X       1111000011110000
            ----------------
            0000111100001111
```

BOOLEAN EXPRESSIONS

The Boolean operators available in Pascal are:

NOT	AND	OR
XOR	=	<
>	<>	<=
>=		

You may also use P <> Q as an exclusive OR function. Since FALSE < TRUE, P <= Q denotes the Boolean operation "P implies Q." Furthermore, the Boolean operators AND and OR are not the same as the WORD and INTEGER operators of the same name that are bitwise logical functions. The Boolean AND and OR operators may or may not evaluate their operations. Consider the following:

```
WHILE (I <= MAX) AND (V [I] <> T) DO I := I + 1;
```

If array V has an upper bound MAX, then the evaluation of V [I]

for I > MAX is a runtime error. This evaluation may or may not take place. Sometimes both operands are evaluated during optimization, and sometimes the evaluation of one may cause the evaluation of the other to be skipped. In the latter case, either operand may be evaluated first.

Alternatively you can use the following construction:

```
WHILE I <= MAX DO
   IF V [I] <> T THEN I := I + 1 ELSE BREAK;
```

The relational operators produce a Boolean result. The types of the operands of a relational operator (except for IN) must be compatible. If they are not compatible, one must be REAL and the other compatible with INTEGER.

Reference types can only be compared with = and <>. To compare an address type with one of the other relational operators, you must use address field notation, as follows:

```
IF (A.R < B.R) THEN <statement>;
```

Except for the string types STRING and LSTRING, you cannot compare files, arrays, and records as wholes. Two STRING types must have the same upper bound to be compared; two LSTRINGs may have different upper bounds.

In LSTRING comparison, characters past the current length are ignored. If the current length of one LSTRING is less than the length of the other and all characters up to the length of the shorter are equal, the compiler assumes the shorter one is "less than" the longer one. However, two LSTRINGs are not considered equal unless all current characters are equal and their current lengths are equal.

The inclusion of special "not-a-number" (NaN) values means that a comparison between two real numbers can have a result other than less-than, equal, or greater-than. The numbers can be unordered, meaning one or both are NaNs. An unordered result is the same as "not equal, not less than, and not greater than."

For example, if variables A or B are NaN values:

1. A < B is false.
2. A <= B is false.
3. A > B is false.
4. A >= B is false.

5. A = B is false.

6. A <> B is, however, true.

REAL comparisons do not follow the same rules as other comparisons in many ways. A < B is not always the same as NOT (B <= A); this prevents some optimizations otherwise done by the compiler. If A is a NaN, then A <> A is true; in fact, this is a good way to check for a NaN value.

SET EXPRESSIONS

Pascal uses several operators in a different way when applied to sets, as follows:

Operator	Meaning in Set Operations
+	Set union
-	Set difference
*	Set intersection
=	Test set equality
<>	Test set inequality
<= and >=	Test subset and superset
< and >	Test proper subset and superset
IN	Test set membership

Any operand whose type is SET OF S, where S is a subrange of T, is treated as if it were SET OF T. (T is restricted to the range from Ø to 255 or the equivalent ORD values.) Either both operands must be PACKED or neither must be PACKED, unless one operand is a constant or constructed set.

With the IN operator, the left operand (an ordinal) must be compatible with the base type of the right operand (a set). The expression X IN B is TRUE if X is a member of the set B, and FALSE otherwise. X can be outside of the range of the base type of B legally. For example, X IN B is always false if the following statements are true:

```
X = 1              {1 is compatible, but not assignment
B = SET OF 2..9    compatible, with 2..9.}
```

"<" and ">" are extended operators, since ISO Pascal does not support them for sets. They test that a set is a proper subset or superset of another set. Proper subsetting does not permit a set as a subset if the two sets are equal.

Expressions involving sets may use the "set constructor," which gives the elements in a set enclosed in square brackets. Each element can be an expression whose type is in the base type of the set or the lower and upper bounds of a range of elements in

the base type. Elements cannot be sets themselves.

Examples of sets involving set constructors:

```
SET_COLOR := [RED, BLUE..PURPLE] - [YELLOW]

SET_NUMBER :=
   [12, J+K, TRUNC (EXP (X))..TRUNC (EXP (X+1))]
```

Set constructor syntax is similar to CASE constant syntax. If X > Y then [X..Y] denotes the empty set. Empty brackets also denote the empty set and are compatible with all sets. Also, if all elements are constant, a set constructor is the same as a set constant.

Like other structured constants, the type identifier for a constant set can be included in a set constant, as in COLORSET [RED..BLUE]. This does not mean that a set constructor with variable elements can be given a type in an expression: NUMBERSET [I..J] is invalid if I or J is a variable.

A set constructor such as [I, J,..K] or an untyped set such as [1, 5..7], is compatible with either a PACKED or an unpacked set. A typed set constant, such as DIGITS [1, 5..7], is only compatible with sets that are PACKED or unpacked, respectively, in the same way as the explicit type of the constant.

FUNCTION DESIGNATORS

A function designator specifies the activation of a function. It consists of the function identifier, followed by a list of "actual parameters" in parentheses:

```
FUNCTION ADD (A, B: INTEGER): INTEGER;   {Declaration of the
                                          function ADD.}
  .
  .
X := ADD (7, X * 4) + 123;               {ADD is function
                                          designator.}
```

These actual parameters substitute, position for position, for their corresponding "formal parameters," defined in the function declaration.

Parameters can be variables, expressions, procedures, or functions. If the parameter list is empty, the parentheses must be omitted. (For more information see chapter 10, Procedures and Functions.

The order of evaluation and binding of the actual parameters varies, depending on the optimizations used. If the $SIMPLE metacommand is on, the order is left to right.

Functions have two different uses:

1. In the mathematical sense, they take one or more values from a domain to produce a resulting value in a range. In this case, if the function never does anything else (such as assign to a global variable or do input/output), it is called a "pure" function.

2. The second type of function may have side effects, such as changing a static variable or a file. Functions of this second kind are said to be "impure."

In ISO Pascal, a function may return either a simple type or a pointer. A pointer returned by a function can only be compared, assigned, or passed as a value parameter. At the extend level, a function can return any assignable type (i.e., any type except a file or super array). The usual selection syntax for reference types, arrays, and records is allowed, following the function designator.

Examples of function designators:

```
SIN (X+Y)

NEXTREC (17) ^ {Here the function return type is a pointer,
                and the returned pointer value is
                dereferenced.}
```

It is more efficient to return a component of a structure than to return a structure and then use only one component of it. The compiler treats a function that returns a structure like a procedure, with an extra VAR parameter representing the result of the function. The function's caller allocates an unseen variable (on the stack) to receive the return value, but this "variable" is only allocated during execution of the statement that contains the function invocation.

EVALUATING EXPRESSIONS

An operator at a higher level is applied before one at a lower level. For instance, the following expression evaluates to 7 and not to 9:

```
1 + 2 * 3
```

You can use parentheses to change operator precedence. Thus, the following evaluates to 9 rather than 7:

```
(1 + 2) * 3
```

If the $SIMPLE is on, sequences of operators of the same precedence are executed from left to right. If the switch is off, the compiler may rearrange expressions and evaluate common subexpressions only once, in order to generate optimized code.

```
X * 3 + 12
```

is an optimization of:

```
3 * (6 + (X - 2))
```

These optimizations may occasionally give you unexpected overflow errors. For example,

```
(I - 100) + (J - 100)
```

will be optimized into the following:

```
(I + J) - 200
```

This may result in an overflow error, although the original expression did not (e.g., if "I" and "J" were each 16400).

An expression in your source file may or may not actually be evaluated when the program runs. For example, the expression F(X + Y)*0 is always zero, so the subexpression (X + Y) and the function call need not be executed.

The compiler does not optimize real expressions as much as, for example, integer expressions, to make sure that the result of a real expression is always what a simple evaluation of the expression, as given, would be. For example, the integer expression

```
((1 + I) - 1) * J
```

is optimized to:

```
I * J
```

but the same expression with real variables is not optimized since the results may be different due to precision loss. Common subexpressions, such as 2 * X in SIN (2 * X) * COS (2 * X), may still be calculated just once and reloaded as necessary, but they are saved in a special 80-bit intermediate precision.

The order of evaluation may be fixed by parentheses:

```
(A + B) + C
```

is evaluated by adding A and B first, but

```
A + B + C
```

may be evaluated by adding A and B, B and C, or even A and C first.

Any expression can be passed as a CONST or CONSTS parameter or have its "address" found. The expression is calculated and stored in a temporary variable on the stack, and the address of this temporary variable can be used as a reference parameter or in some other address context.

To avoid ambiguities, enclose such an expression with operators or function calls in parentheses. For example, to invoke a procedure FOO (CONST X, Y: INTEGER), FOO (I, (J+14)) must be used instead of FOO (I, J+14). This implies a subtle distinction in the case of functions. For example:

```
FUNCTION SUM (CONST A, B: INTEGER): INTEGER;
BEGIN
  SUM := A;
  IF B <> Ø THEN
    SUM := SUM (SUM, (SUM (B, Ø) - 1)) + 1;
END;
```

In this example, SUM is called recursively subtracting one from B until B is zero.

The use of a function identifier in a WITH statement follows a similar rule. For example, given a parameterless function, COMPLEX, which returns a record, "WITH COMPLEX" means "WITH the current value of the function." This can only occur inside the COMPLEX function itself. However, "WITH (COMPLEX)" causes the function to be called and the result assigned to a temporary local variable.

Another way to describe this is to distinguish between "address" and "value" phrases. The left-hand side of an assignment, a reference parameter, the ADR and ADS operators, and the WITH statement all need an address. The right hand side of an assignment and a value parameter all need a value.

If an address is needed but only a value is available, the value must be put into memory so it has an address. For constants, the value goes in static memory; for expressions, the value goes in stack (local) memory. A function identifier refers to the current value of the function as an address, but causes the function to be called as a value.

In the scope of a function, the intrinsic procedure RESULT permits a reference to the current value of a function instead of invoking it recursively. For a function F, this means ADR F and ADR RESULT (F) are the same: the address of the current value of F. RESULT forces use of the current value in the same way that putting the function in parentheses, as in (F(X)), forces

evaluation of the function.

OTHER FEATURES OF EXPRESSIONS

EVAL and RESULT are two procedures available for use with expressions. EVAL obtains the effect of a procedure from a function; RESULT yields the current value of a function within a function or nested procedure or function. The function RETYPE allows you to change the type of a value.

The EVAL Procedure

EVAL evaluates its parameters without actually calling anything. Generally, you use EVAL to obtain the effect of a procedure from a function. In such cases, the values returned by functions are of no interest, so EVAL is only useful for functions with side effects. For example, a function that advances to the next item and also returns the item might be called in EVAL just to advance to the next item, since there is no need to obtain a function return value.

Examples of the EVAL procedure:

```
EVAL (NEXTLABEL (TRUE))
EVAL (SIDEFUNC (X, Y), INDEX (4), COUNT)
```

The RESULT Function

Within the scope of a function, the intrinsic procedure RESULT permits a reference to the current value of a function instead of invoking it recursively. For a function F, this means ADR F and ADR RESULT (F) are the same; that is, the address of the current value of F. RESULT forces use of the current value in the same way that putting the function in parentheses as in (F (X)) forces evaluation of the function.

Examples of the RESULT function:

```
FUNCTION FACTORIAL (I: INTEGER): INTEGER;
BEGIN
  FACTORIAL := 1;
  WHILE I > 1 DO
  BEGIN
    FACTORIAL := I * RESULT (FACTORIAL);
    I := I - 1;
  END;
END;

FUNCTION ABSVAL (I: INTEGER): INTEGER;
BEGIN
  ABSVAL := I;
```

```
  IF I < Ø THEN ABSVAL := -RESULT (ABSVAL);
END;
```

The RETYPE Function

You can change the type of a value by using the RETYPE function. If the new type is a structure, RETYPE can be followed by the usual selection syntax. You must be cautious in using RETYPE since it works on the memory byte level and ignores whether the low order byte of a two-byte number comes first or second in memory.

Examples of the RETYPE function:

```
RETYPE (COLOR, 3)              {inverse of ORD}
RETYPE (STRING2, I*J+K) [2]    {effect may vary}
```

CHAPTER 9
STATEMENTS

CONTENTS

The body of a program, procedure, or function contains statements. Statements denote actions that the program can execute. There are two types of statements, simple and structured. A simple statement has no parts that are themselves other statements; a structured statement consists of two or more other statements.

STATEMENT SYNTAX

Pascal statements are separated by a semicolon (;) and enclosed by reserved words such as BEGIN and END. A statement begins, optionally, with a label. Each of these three elements of statement syntax are discussed below:

Labels

Any statement referred to by a GOTO statement must have a label. In standard Pascal a label consists of one or more digits; leading zeros are ignored. Constant identifiers, expressions, and nondecimal notation cannot serve as labels. In this extended Pascal a label can also be an identifier. All labels must be declared in a LABEL section.

Example using labels and GOTO statements:

```
PROGRAM LOOPS(INPUT,OUTPUT);
LABEL 1, HAWAII, MAINLAND;

BEGIN
  MAINLAND: GOTO 1;
  HAWAII: WRITELN ('Here I am in Hawaii');
  1: GOTO HAWAII
END.
```

A loop label is any label immediately preceding a looping statement; WHILE, REPEAT, FOR, BREAK or CYCLE statement all refer to a loop label.

Both a CASE constant list and a GOTO label may precede a statement, in which case the CASE constants come first and then the GOTO label. In the following example, 321 is a CASE value, 123 is label:

```
321:  123:  IF LOOP THEN GOTO 123
```

Statements Separation

Semicolons separate statements. Semicolons do not terminate statements. For example, the following statements are separated by semicolons:

```
BEGIN
   1Ø: WRITELN;
   A := 2 + 3;
   GOTO 1Ø
END
```

A common error is to terminate the THEN clause in an IF/THEN/ELSE statement with a semicolon. Thus, the following example generates a warning message:

```
IF A = 2 THEN WRITELN;
ELSE
IF A = 3
```

Another common error is to put a semicolon after the DO in a WHILE or FOR statement:

```
FOR I := 1 TO 1Ø DO;
BEGIN
  A[I] := I;
  B[I] := 1Ø - I;
END;
```

The above example will "execute" an empty ten times, then execute the array assignments once. Since there are occasional legitimate uses for repeating an empty statement, no warning is given when this occurs. The semicolon also follows the reserved word END at the close of a block of program statements.

Begin/End

Whenever you want a program to execute a group of statements, you may enclose the block with the reserved words BEGIN and END. For example, the following group of statements between BEGIN and END will all be executed if the condition in the IF statement is TRUE:

```
IF (MAX > 1Ø) THEN
BEGIN
  MAX = 1Ø;
  MIN = Ø;
  WRITELN (MAX,MIN)
END;
WRITELN ('done')
```

You can also substitute a pair of square brackets for the pair of keywords BEGIN and END.

SIMPLE STATEMENTS

A simple statement is one in which no part constitutes another statement. Simple statements are as follows:

1. the assignment statement

2. the procedure statement

3. the GOTO statement

4. the empty statement

5. the BREAK, CYCLE and RETURN statements.

The empty statement contains no symbols and denotes no action. It is included in the definition of the language primarily to permit you to use a semicolon after the last in a group of statements enclosed between BEGIN and END.

Assignment Statements

The assignment statement replaces the current value of a variable with a new value, which you specify as an expression. Assignment is denoted by an adjacent colon and equal sign characters (:=).

Examples of assignment statements:

```
A := B

A[I] := 12 * 4 + (B * C)

A := ADD (1,1)
```

The value of the expression must be assignment compatible with the type of the variable. Selection of the variable may involve indexing an array or dereferencing a pointer or address. If it does, the compiler may, depending on the optimizations performed, mix these actions with the evaluation of the expression. If the $SIMPLE metacommand is on, the expression is evaluated first.

Within the block of a function, an assignment to the identifier of the function sets the value returned by the function. The assignment to a function identifier may occur either within the actual body of the function or in the body of a procedure or function nested within it.

If the $RANGECK is on, an assignment to a set, subrange, or LSTRING variable may imply a runtime call to the error checking code.

The optimizer allows each section of code without a label or other point that could receive control to be eligible for rearrangement and common subexpression elimination. Naturally, the order of execution is retained when necessary.

Given these statements,

```
X := A + C + B;
Y := A + B;
Z := A
```

the compiler might generate code to perform the following operations:

1. Get the value of A and save it.

2. Add the value of B and save the result.

3. Add the value of C and assign it to X.

4. Assign the saved A + B value to Y.

5. Assign the saved A value to Z.

This optimization occurs only if assignment to X and Y and getting the value of A, B, or C are all independent. If C is a function without the PURE attribute and A is a global variable, evaluating C might change A. Then since the order of evaluation within an expression in this case is not fixed, the value of A in the first assignment could be the old value or the new one.

However, since the order of evaluation among statements is fixed, the value of A in the second and third assignments is the new value. The following actions may limit the ability of the optimizer to find common subexpressions:

1. assignment to a nonlocal variable

2. assignment to a reference parameter

3. assignment to the referent of a pointer

4. assignment to the referent of an address variable

5. calling a procedure

6. calling a function without the PURE attribute

The optimizer does allow a single variable with two identifiers, perhaps one as a global variable and one as a reference parameter.

Procedure Statements

A procedure statement executes the procedure denoted by the procedure identifier. For example:

```
PROCEDURE DO_IT;
BEGIN
```

```
  WRITELN('Did it')
END;
```

DO_IT is now a statement that can be executed simply by invoking its name:

```
DO_IT
```

If you declare the procedure with a formal parameter list, the procedure statement must include the actual parameters. Predeclared procedures are also available. One of the predeclared procedures is ASSIGN. You need not declare in order to use it. For more information see Chapter 11, Available Procedures and Functions.

```
ASSIGN (INFILE, 'MYFILE')
```

Note that the ASSIGN procedure contains a parameter list. These parameters are the actual parameters that are bound to the formal parameters in the procedure declaration.

The GOTO Statement

A GOTO statement indicates that further processing continues at another part of the program text, namely at the place of the label. You must declare a LABEL in a LABEL declaration section, before using it in a GOTO statement. The following restrictions apply to the use of GOTO statements:

1. A GOTO must not jump to a more deeply nested statement, that is, into an IF, CASE, WHILE, REPEAT, FOR, or WITH statement. GOTOs from one branch of an IF or CASE statement to another are permitted.

2. A GOTO from one procedure or function to a label in the main program or in a higher level procedure or function is permitted. A GOTO may jump out of one of these statements, so long as the statement is directly within the body of the procedure or function. However, such a jump generates extra code both at the location of the GOTO and at the location of the label. The GOTO and label must be in the same compiland, since labels, unlike variables, cannot be given the PUBLIC attribute.

If the $GOTO metacommand is on, every GOTO statement is flagged with a warning that reminds you that "GOTOs are considered harmful." This may be useful either in an educational environment or for finding all GOTOs in a program in order to locate a bug. The J (jumps) column of the listing file contains the following:

1. A plus (+) or an asterisk (*) flags a GOTO to a label later in the listing.

2. A minus sign (-) or an asterisk (*) marks a GOTO to a label already encountered in the listing.

The BREAK, CYCLE, and RETURN Statements

The BREAK, CYCLE, and RETURN statements are allowed in addition to the simple statements already described. These statements perform the following functions:

1. BREAK exits the currently executing loop.
2. CYCLE exits the current iteration of a loop and starts the next iteration.
3. RETURN exits the current procedure, function, program, or implementation.

All three statements are functionally equivalent to a GOTO statement.

1. A BREAK statement is a GOTO to the first statement after a repetitive statement.
2. A CYCLE statement is a GOTO to an implied empty statement after the body of a repetitive statement. This jump starts the next iteration of a loop. In either a WHILE or REPEAT statement, CYCLE performs the Boolean test in the WHILE or UNTIL clause before executing the statement again; in a FOR statement, CYCLE goes to the next value of the control variable.
3. A RETURN statement is a GOTO to an implied empty statement after the last statement in the current procedure or function or the body of a program or implementation.

The J (jump) column in the listing file contains a plus sign (+) or an asterisk (*) for a BREAK statement, a minus sign (-) or an asterisk (*) for a CYCLE statement, and an asterisk (*) for a RETURN statement. For more information see the Listing File Format under Chapter 14, Compiling, linking, and Executing Programs.

BREAK and CYCLE have two forms, one with a loop label and one without. If you give a loop label, the label identifies the loop to exit or restart. If you don't give a label, the innermost loop is assumed, as shown in the following example:

```
OUTER: FOR I := 1 TO N1 DO
  INNER: FOR J := 1 TO N2 DO
```

```
IF A [I, J] = TARGET THEN BREAK OUTER;
```

STRUCTURED STATEMENTS

Structured statements are themselves composed of other statements. There are four kinds of structured statements:

1. compound statements
2. conditional statements
3. repetitive statements
4. WITH statement

The control level is shown in the the C (control) column of the listing file. The value in the C column is incremented each time control passes to a nested statement; conversely, this value is decremented each time control passes back to the nesting statement. This helps you search for a missing or extra END in a program.

Compound Statements

The compound statement is a sequence of simple statements, enclosed by the reserved words BEGIN and END. The components of a compound statement execute in the same sequence as they appear in the source file.

Examples of compound statements:

```
BEGIN
   TEMP := A [I];
   A[I] := A [J];
   A [J] := TEMP
END

BEGIN
   OPEN_DOOR;
   LET_EM_IN;
   CLOSE_DOOR;
END
```

All conditional and repetitive control structures (except REPEAT) operate on a single statement, not on multiple statements with ending delimiters. You may substitute a pair of square brackets for the BEGIN and END pair of reserved words. Note that a right bracket (]) matches only a left bracket ([) (not a BEGIN, CASE, or RECORD). In other words, right bracket is not a synonym for END.

Brackets may not be used as synonyms for BEGIN and END to enclose the body of a program, implementation, procedure, or function; only BEGIN and END can be used for this purpose.

Examples of brackets replacing BEGIN and END:

```
IF FLAG THEN
   [X := 1; Y := -1]
ELSE
   [X := -1; Y := Ø];

WHILE P.N <> NIL DO
   [Q := P;  P := P.N;  DISPOSE (Q)];
```

Conditional Statements

A conditional statement selects for execution only one of its component statements. The conditional statements are the IF and CASE statements. You should use the IF statement for one or two conditions, the CASE statement for multiple conditions.

The IF Statement

The IF statement allows for conditional execution of a statement. If the Boolean expression following the IF is true, the statement following the THEN is executed. If the Boolean expression following the IF is false, the statement following the ELSE, if present, is executed.

Examples of IF statements:

```
IF I > Ø THEN
   I := I - 1
ELSE I := I + 1

IF (I <= TOP) AND (ARRI [I] <> TARGET) THEN
   I := I + 1

IF I <= TOP THEN
   IF ARRI [I] <> TARGET THEN
      I := I + 1

IF I = 1 THEN
   IF J = 1 THEN
      WRITELN('I equals J')
   ELSE
      WRITELN('DONE only if I = 1 and J <> 1')
                {This ELSE is paired with the most deeply
                 nested IF.  Thus, the second WRITELN is
                 executed only if I = 1 and J <> 1.}

IF I = 1 THEN BEGIN
```

```
      IF J = 1 THEN WRITELN('I equals J')
      END
   ELSE
      WRITELN('DONE only if I <> 1')
                  {Now the ELSE is paired with the first IF,
                   since the second IF statement is
                   bracketed by the BEGIN/END pair.  Thus,
                   the second WRITELN is executed if I <> 1.}
```

A semicolon (;) preceding an ELSE is always incorrect. The compiler skips it during compilation and issues a warning message.

The CASE Statement

The CASE statement consists of an expression (called the CASE index) and a list of statements. Each statement is preceded by a constant list, called a CASE constant list. The one statement executed is the one whose CASE constant list contains the current value of the CASE index. The CASE index and all constants must be of compatible, ordinal types.

Examples of CASE statements:

```
   CASE OPERATOR OF
     PLUS:  X := X + Y;
     MINUS: X := X - Y;
     TIMES: X := X * Y
   END                    {OPERATOR is the CASE index.  PLUS,
                          MINUS, and TIMES are CASE constants.}

   CASE NEXTCH OF
     'A'..'Z', '_'        : IDENTIFIER;
     '+', '-', '*', '/'   : OPERATOR;
     OTHERWISE
       WRITE ('Unknown Character')
   END
```

The CASE constant syntax is the same as for RECORD variant declarations. In standard Pascal, a CASE constant is one or more constants separated by commas. With this extended Pascal you can substitute a range of constants, such as 'A'..'Z', for a constant. No constant value can apply to more than one statement.

The CASE statement can also be ended with an OTHERWISE clause. The OTHERWISE clause contains additional statements to be executed in the event that the CASE index value is not in the given set of CASE constant values. Note that OTHERWISE cannot be used with RECORD declarations. If the CASE index value is not in the set and no OTHERWISE clause is present one of the of two things happen:

1. If the $RANGECK is on, a runtime error is generated.

2. If the $RANGECK is off, the result is undefined.

Depending on optimization, the code generated by the compiler for a CASE statement may be either a "jump table" or series of comparisons (or both). If it is a jump table, a jump to an arbitrary location in memory can occur if the control variable is out of range and the range checking switch is off.

Repetitive Statements

Repetitive statements specify repeated execution of a statement. These statements are functionally equivalent to a GOTO but easier to use.

The WHILE Statement

The WHILE statement repeats a statement zero or more times, until a Boolean expression becomes false.

Examples of WHILE statements:

```
WHILE P <> NIL DO P := NEXT (P)

WHILE NOT MICKEY DO
   BEGIN
      NEXTMOUSE;
      MICE := MICE + 1
   END
```

The WHILE statement should be used when no iterations of the loop are desired. The REPEAT statement should be used to execute loops where at least one iteration of the the loop is desired.

The REPEAT Statement

The REPEAT statement repeats a sequence of statements one or more times, until a Boolean expression becomes true.

Examples of REPEAT statements:

```
REPEAT
   READ (LINEBUFF);
   COUNT := COUNT + 1
UNTIL EOF;

REPEAT GAME UNTIL TIRED;
```

You should use the REPEAT statement to execute statements, one or more times until a condition becomes true. This differs from the WHILE statement in which a single statement may not be executed at all.

The FOR Statement

The FOR statement indicates to the compiler to execute a statement repeatedly while a progression of values is assigned to a variable, called the control variable of the FOR statement. The values assigned start with a value called the initial value and end with one called the final value.

The FOR statement has two forms, one where the control variable increases in value and one where the control variable decreases in value:

```
FOR I := 1 TO 1Ø DO              {I is the control variable.}
  SUM := SUM + VICTORVECTOR [I];

FOR CH := 'Z' DOWNTO 'A' DO      {CH is the control variable.}
  WRITE (CH);
```

You can also use a FOR statement to step through the values of a set, as follows:

```
FOR TINT := LOWER (SHADES) TO UPPER (SHADES) DO
      IF TINT IN SHADES
      THEN PAINT_AREA (TINT);
```

The following are explicit rules defined within ISO Pascal regarding the control variables in FOR statements:

1. It must be of an ordinal type.

2. It must also be an entire variable, not a component of a structure.

3. It must be local to the immediately enclosing program, procedure, or function and cannot be a reference parameter of the procedure or function.

   However, in this extended Pascal, the control variable may also be any STATIC variable, such as a variable declared at the program level, unless the variable has a segmented ORIGIN attribute.

4. No assignments to the control variable are allowed in the repeated statement. This error is caught by making the control variable READONLY within the FOR statement; it is not caught when a procedure or function invoked by

the repeated statement alters the control variable. The control variable cannot be passed as a VAR (or VARS) parameter to a procedure or function.

5. The initial and final values of the control variable must be compatible with the type of the control variable. If the statement is executed, both the initial and final values must also be assignment compatible with the control variable. The initial value is always evaluated first, and then the final value. Both are evaluated only once before the statement executes.

The statement following the DO is not executed at all if:

1. The initial value is greater than the final value in the TO case.

2. The initial value is less than the final value in the DOWNTO case.

The sequence of values given the control variable starts with the initial value. This sequence is defined with the SUCC function for the TO case or the PRED function for the DOWNTO case until the last execution of the statement, when the control variable has its final value.

The value of the control variable, after a FOR statement terminates naturally (whether or not the body executes), is undefined. It may vary due to optimization and, if $INITCK is on, may be set to an uninitialized value. However, the value of the control variable after leaving a FOR statement with GOTO or BREAK is defined as the value it had at the time of exit.

At the standard level, the body of a FOR statement may or may not be executed, so a test is necessary to see whether the body should be executed at all. However, if the control variable is of type WORD (or a subrange) and its initial value is a constant zero, the body must be executed no matter what the final value. In this case, no extra test need be executed and no code is generated to perform such a test.

You may use temporary control variables:

```
FOR VAR control-variable
```

The prefix VAR causes the control variable to be declared local to the FOR statement (i.e., at a lower scope) and need not be declared in a VAR section. Such a control variable is not available outside the FOR statement, and any other variable with the same identifier is not available within the FOR statement itself. Other synonymous variables are, however, available to procedures or functions called within the FOR statement.

Examples of temporary control variables:

```
FOR VAR I := 1 TO 100 DO
   SUM := SUM + VICTOR [I]

FOR VAR COUNTDOWN := 10 DOWNTO LIFT_OFF DO
   MONITOR_ROCKET
```

The BREAK and CYCLE Statements

In theory, a program using the BREAK and CYCLE statements does not need to use any GOTO statements. Each of these two statements has two forms, one with a loop label and one without. A loop label is a normal GOTO label prefixed to a FOR, WHILE, or REPEAT statement. You should use integers for labels referenced by GOTOs and identifiers for loop labels.

Examples of CYCLE and BREAK statements:

```
LABEL SEARCH, CLIMB;
  .
  .
SEARCH: WHILE I <= I_TOP DO
  IF PILE [I] = TARGET THEN BREAK SEARCH
  ELSE I := I + 1;
  .
  .
FOR I := 1 TO N DO
  IF NEXT [I] = NIL THEN BREAK;
  .
  .
CLIMB:  WHILE NOT ITEM^.LEAF DO
  BEGIN
    IF ITEM^.LEFT <> NIL
      THEN [ITEM := ITEM^.LEFT;  CYCLE CLIMB];
    IF ITEM^.RIGHT <> NIL
      THEN [ITEM := ITEM^.RIGHT; CYCLE CLIMB];
    WRITELN ('Very strange node');
    BREAK CLIMB
  END;
```

The WITH Statement

The WITH statement opens the scope of a statement to include the fields of one or more records, so you can refer to the fields directly. For example, the following statements are equivalent:

```
WITH PERSON DO WRITE (NAME, ADDRESS, PHONE)
WRITE (PERSON.NAME, PERSON.ADDRESS, PERSON.PHONE)
```

The record given may be a variable, constant identifier,

structured constant, or function identifier; it may not be a component of a PACKED structure. If you use a function identifier, it refers to the function's local result variable. If the record given in a WITH statement is a file buffer variable, the compiler issues a warning, since changing the position in the WITH statement may cause an error.

The record given can also be any expression in parentheses, in which case the expression is evaluated and the result assigned to a temporary (hidden) variable. If you want to evaluate a function designator, you must enclose it in parentheses.

You can give a list of records after the WITH, separated by commas. Each record must be of a different type from all the others, since the field identifiers refer only to the last instance of the record with the type. These statements are equivalent:

```
WITH PMODE, QMODE DO statement
WITH PMODE DO WITH QMODE DO statement
```

Any record variable of a WITH statement that is a component of another variable is selected before the statement is executed. Active WITH variables should not be passed as VAR or VARS parameters, nor can their pointers be passed to the DISPOSE procedure. However, these errors are not caught by the compiler. Assignments to any of the record variables in the WITH list or components of these variables are allowed, as long as the WITH record is a variable.

Every WITH statement allocates an address variable that holds the address of the record. If the record variable is on the heap, the pointer to it should not be DISPOSEd within the WITH statement. If the record variable is a file buffer, no I/O should be done to the file within the WITH statement.

Sequential Control

To increase execution speed or to ensure correct evaluation, it is often useful in IF, WHILE, and REPEAT statements to treat the Boolean expression as a series of tests. If one test fails, the remaining tests are not executed. The sequential control operators provide for the following tests:

1. AND THEN

   X AND THEN Y is false if X is false; Y is only evaluated if X is true.

2. OR ELSE

   OR ELSE Y is true if X is true; Y is only evaluated if X is false.

If you use several sequential control operators, the compiler evaluates them strictly from left to right. You may only include these operators in the Boolean expression of an IF, WHILE, or UNTIL clause; they may not be used in other Boolean expressions. Furthermore, they may not occur in parentheses and are evaluated after all other operators.

Examples of sequential control operators:

```
IF SYM <> NIL AND THEN SYM^.VAL < Ø THEN
   NEXT_SYMBOL

WHILE I <= MAX AND THEN VECT [I] <> KEY DO
   I := I + 1;

REPEAT GEN (VAL)
UNTIL VAL = Ø OR ELSE (QU DIV VAL) = Ø;

WHILE POOR AND THEN GETTING_POORER
   OR ELSE BROKE AND THEN BANKRUPT DO
      GET_RICH
```

CHAPTER 10
PROCEDURES AND FUNCTIONS

CONTENTS

Procedures and functions act as subprograms that execute under the supervision of a main program. Unlike programs, however, procedures and functions can be nested within each other and can even call themselves. Furthermore, they have sophisticated parameter passing capabilities that programs lack.

Procedures are invoked as program statements; functions can be invoked in program statements wherever a value is called for. The general format for procedures and functions is similar to the format for programs. The format includes a heading, declarations, and a body.

Example of a procedure declaration:

```
PROCEDURE MODEL (I: INTEGER; R: REAL);                {Heading}


LABEL 123;                    {Beginning of declaration section}
CONST ATOP = 199;
TYPE  INDEX = Ø..ATOP;
VAR   ARAY: ARRAY [INDEX] OF REAL;
      J: INDEX;


FUNCTION FONE (RX: REAL): REAL;          {Function declaration}
BEGIN
   FONE := RX * I
END;


PROCEDURE FOUT (RY: REAL);               {Procedure declaration}
BEGIN
   WRITE ('Output is ', RY)
END;


BEGIN                                    {Body of procedure MODEL}
  FOR J := Ø TO ATOP DO
    IF GLOBALVAR THEN
         FOUT (FONE (R + ARAY [J]))      {Activation of procedure
                                          FOUT with value return-
                                          by function FONE
      ELSE
         GOTO 123;
  123:  WRITELN ('Done');
END;
```

The declaration of a procedure or function associates an identifier with a portion of a program. Later, you can activate that portion of the program with the appropriate procedure statement or function designator.

PROCEDURES

The above example illustrates the general format of a procedure declaration. The heading is followed by:

1. declarations for labels, constants, types, variables, and values
2. local procedures and functions
3. the body, which is enclosed by the reserved words BEGIN and END

When the body of a procedure finishes execution, control returns to the program element that called it. At the standard level, the order of declarations must be as follows:

1. LABEL
2. CONST
3. TYPE
4. VAR
5. procedures and functions

At the extended level, you can have any number of LABEL, CONST, TYPE, VAR, and VALUE sections, as well as procedure and function declarations, in any order. However, putting variable declarations after procedure and function declarations guarantees that these variables will not be used by any of the procedures or functions.

In general, the initial values of variables are not defined. The VALUE section, which should follow the VAR section, lets you explicitly initialize program, module, implementation, STATIC, and PUBLIC variables. If the initialization switch ($INITCK) is on, all INTEGER, INTEGER subrange, REAL, and pointer variables are set to an uninitialized value. File variables are always initialized, regardless of the setting of the initialization switch.

FUNCTIONS

Functions are the same as procedures, except that they are invoked in an expression instead of a statement and they return a value.

Function declarations define the parts of a program that compute a value. Functions are activated when a function designator, which is part of an expression, is evaluated.

A function declaration has the same format as a procedure declaration, except that the heading also gives the type of value returned by the function.

Example of a function heading:

```
FUNCTION MAXIMUM (I, J:  INTEGER):  INTEGER;
```

Within the block of a function, either in the body itself or in a procedure or function nested within the block, at least one assignment to the function identifier must be executed to set the return value. The compiler doesn't check for this assignment at runtime, unless the initialization switch is on and the returned type is INTEGER, REAL, or a pointer. However, if there is no assignment at all to the function identifier, the compiler issues an error message.

At the standard level, functions can return any simple type (ordinal, REAL, or INTEGER4) or a pointer. At the extended level, functions can return any simple, structured, or reference type. However, they cannot return any type that cannot be assigned (i.e., a super array type or a structure containing a file, although a super array derived type is permitted).

A function identifier in an expression invokes the function recursively, rather than giving the current value of the function.

To obtain the current value, you must use the function RESULT, which is available at the extended level. This function takes the function identifier as a parameter. The following is an example of RESULT function used to obtain the current value of a function within an expression:

```
FUNCTION FACT (F: REAL): REAL;
BEGIN
  FACT := 1;
  WHILE F > 1 DO
     BEGIN
       FACT := RESULT (FACT) * F;
       F := F-1
     END
END
```

Using the RESULT function is more efficient than using a separate local variable for the value of the function and then assigning this local variable to the function identifier before returning. If the function has a structured value, the usual component selection syntax can follow the RESULT function.

A function identifier on the left side of an assignment refers to the function's local variable, which contains its current value, instead of invoking the function recursively. Other places where using the function identifier refers to this local variable are

the following:

1. a reference parameter

2. the record of a WITH statement

3. the operand of an ADR or ADS operator

All of these uses involve getting the address (not the value) of a variable.

Instead of using the function's local variable, you may want to invoke the function and use the return value. Getting the address of an expression involves evaluating the expression, putting the resulting value into a temporary (hidden) variable, and using the address of this variable. To do this for a function, you must force evaluation by putting the function designator in parentheses, as follows:

```
TYPE IREC = RECORD
              I: INTEGER;
     END;

FUNCTION SUM (A, B: INTEGER): IREC; {Return sum of A and B.}

BEGIN
  IF TUESDAY THEN                    {On Tuesdays, we recurse.}
  BEGIN
    IF B = Ø THEN BEGIN SUM := A; RETURN END;
    WITH (SUM (A,B-1))               {Call SUM recursively.}
      DO SUM.I := I + 1              {I is result of call.}
  END
  ELSE
    WITH SUM
      DO I := A + B;                 {I is local variable.}
END;
```

ATTRIBUTES AND DIRECTIVES

An attribute gives additional information about a procedure or function. Attributes are available at the extended level of Pascal. They are placed after the heading, enclosed in brackets and separated by commas. Available attributes include ORIGIN, PUBLIC, and PURE.

A directive gives information about a procedure or function, but it also indicates that only the heading of the procedure or function occurs, by replacing the block (declarations and body) normally included after the heading.

EXTERN and FORWARD are the only directives available. EXTERN can only be used with procedures or functions directly nested in a

program, module, implementation, or interface. This restriction prevents access to nonlocal stack variables. The following atttributes and directives apply to procedures and functions:

Name	Purpose
FORWARD	A directive. Lets you call a procedure or function before you give its block in the source file.
EXTERN	A directive. Indicates that a procedure or function resides in another loaded module.
PUBLIC	An attribute. Indicates that a procedure or function may be accessed by other loaded modules.
ORIGIN	An attribute. Tells the compiler where the code for an EXTERN procedure or function resides.
PURE	An attribute. Signifies that the function does not modify any global variables.

The following rules apply when you combine attributes in the declaration of procedures and functions:

1. Any function may be given the PURE attribute.

2. Procedures and functions with attributes must be nested directly within a program, module, or unit. The only exception to this rule is the PURE attribute.

3. PUBLIC and EXTERN are mutually exclusive, as are PUBLIC and ORIGIN.

The EXTERN or FORWARD directive is given automatically to all constituents of the interface of a unit; in the implementation, PUBLIC is given automatically to all constituents that are not EXTERN.

Since you declare the constituents of a unit only in the interface (not in the implementation), the interface is where you give the attributes. You may give the EXTERN directive in an implementation by declaring all EXTERN procedures and functions first; you may not use ORIGIN in either the interface or implementation of a unit.

In a module, you may give a group of attributes in the heading to apply to all directly nested procedures and functions. The only exception to this rule is the ORIGIN attribute, which may apply only to a single procedure or function.

If the PUBLIC attribute is one of a group of attributes in the heading of a module, an EXTERN attribute given to a procedure or function within the module explicitly overrides the global PUBLIC

attribute. If the module heading has no attribute clause, the PUBLIC attribute is assumed for all directly nested procedures and functions.

The PUBLIC attribute allows a procedure or function to be called by other loaded code, and cannot be used with the EXTERN directive. The EXTERN directive permits a call to some other loaded code, using either the ORIGIN address or the linker. PUBLIC, EXTERN, and ORIGIN provide a low level way to link Pascal routines with other routines in Pascal or other languages.

A procedure or function declaration with the EXTERN or FORWARD directive consists only of the heading, without an enclosed block. EXTERN routines have an implied block outside of the program. FORWARD routines are fully declared (have a block) later in the same compiland. Both directives are available at the standard level. The keyword EXTERNAL is a synonym for EXTERN. The PURE attribute applies only to functions, not to procedures.

The FORWARD Directive

A FORWARD declaration allows you to call a procedure or function before you fully declare it in the source text. This permits indirect recursion, where A calls B and B calls A. You can make a FORWARD declaration by specifying a procedure or function heading, followed by the directive FORWARD. Later, you can actually declare the procedure or function, without repeating the formal parameter list or any attributes or the return type a function.

Example of a FORWARD declaration:

```
FUNCTION ALPHA (Q, R: REAL): REAL [PUBLIC];  FORWARD;


PROCEDURE BETA (VAR S, T: REAL);    {Call for ALPHA}
BEGIN
  T := ALPHA (S, 3.14)
END;


FUNCTION ALPHA;  {Actual declaration of ALPHA,
BEGIN             without parameter list}
  ALPHA := (Q + R);
  IF R < Ø.Ø THEN BETA (3.14,ALPHA);
END;
```

The EXTERN Directive

The EXTERN directive identifies a procedure or function that resides in another loaded module. You give only the heading of

the procedure or function, followed by the word EXTERN. The actual implementation of the procedure or function is presumed to exist in some other module.

EXTERN is an attribute when used with a variable, but a directive when used with a procedure or function. The EXTERN directive for a particular procedure or function within a module overrides the PUBLIC attribute given for the entire module. The EXTERN directive is also permitted in an implementation of a unit for a constituent procedure or function.

All such external constituents must be declared at the beginning of the implementation, before all other procedures and functions. Any procedure or function with the EXTERN directive must be directly nested within a program.

Examples of procedure and function headings with EXTERN directive:

```
FUNCTION POWER (X, Y:  REAL):  REAL;  EXTERN;

PROCEDURE ACCESS (KEY:  KTYP) [ORIGIN SYSB+4];
EXTERN;
```

In the above examples, the function POWER is declared EXTERN, as is the procedure ACCESS. Both are implemented in external compilands. ACCESS also has the ORIGIN attribute. Note that when a procedure or a function is declared EXTERN, it cannot have been already declared forward.

The PUBLIC Attribute

The PUBLIC attribute indicates a procedure or function that you can access from other loaded modules. In general, you access PUBLIC procedures and functions from other loaded modules by declaring them EXTERN in the modules that call them. Thus, you can declare a procedure PUBLIC and define it in one module, and use it in another simply by declaring it EXTERN in the other module.

As with variables, the identifier of the procedure or function is passed to the linker, where it may be truncated if the linker requires it. PUBLIC and ORIGIN are mutually exclusive; PUBLIC routines need a following block, and ORIGIN routines must be EXTERN.

Any procedure or function with the PUBLIC attribute must be directly nested within a program or implementation. Linkage between Pascal routines can be done with separately compiled units, discussed in Chapter 13, Compilands.

Examples of procedures and functions declared PUBLIC:

```
FUNCTION POWER (X, Y: REAL): REAL [PUBLIC];{PUBLIC indicates
 BEGIN                                      that function
    .                                       POWER is availab-
    .                                       le to other modu-
 END;                                       les.}

PROCEDURE ACCESS (KEY: KTYP) [ORIGIN SYSB+4, PUBLIC];
 BEGIN
   .              {Invalid since ORIGIN must also be EXTERN.}
   .
 END;
```

The ORIGIN Attribute

The ORIGIN attribute must be used with the EXTERN directive; ORIGIN indicates to the compiler the location of the procedure or function, so that the linker does not require a corresponding PUBLIC identifier. For example:

```
FUNCTION A_TO_D (C: SINT): SINT [ORIGIN #100];   EXTERN;
```

In the above example, the function A_TO_D takes a SINT value as a parameter (SINT is the predeclared integer subrange from -127 to +127). The function is located at the hexadecimal address 100.

ORIGIN always implies EXTERN. Thus, procedures or functions that have previously been declared FORWARD cannot be declared with the ORIGIN attribute. This also means that ORIGIN cannot be given as an attribute after the module heading.

The ORIGIN attribute cannot be used with a constituent of a unit, either in an interface or in an implementation. As with variables, the origin can be a segmented address. A nonsegmented procedural origin assumes the current code segment with the offset given with the attribute.

The PURE Attribute

The PURE attribute applies only to functions, not to procedures or variables. PURE indicates to the compiler's optimizer that the function does not modify any global variables either directly or by calling some other procedure or function.

Example of a PURE declaration:

```
FUNCTION AVERAGE (CONST TABLE:  RVECTOR): REAL [PURE];
```

As an illustration, examine these statements:

```
A := VEC [I * 10 + 7];
B := FOO;
C := VEC [I * 10 + 9]
```

If the function FOO is given the PURE attribute, the optimizer only generates code to compute I*1Ø once. However, FOO, if it is not declared PURE, may modify I so that I*1Ø must be recomputed after the call to FOO.

Functions are not considered PURE unless given the attribute explicitly. A PURE function cannot do the following:

1. assign to a nonlocal variable
2. use the value of a global variable
3. have any VAR or VARS parameters (CONST and CONSTS parameters are permitted)
4. modify the referents of references passed by value (e.g., pointer or address type referents)
5. call any functions that are not PURE
6. do input or output

Since the result of a PURE function with the same parameters must always be the same, the entire function call may be optimized away. For example, if in the following statements DSIN is PURE, the compiler only calls DSIN once:

```
HX := A * DSIN (P[I, J] * 2);
HY := B * DSIN (P[I, J] * 2);
```

PROCEDURE AND FUNCTION PARAMETERS

Procedures and functions may take three different type of parameters:

1. value parameters
2. reference parameters
3. procedural and functional parameters

A formal parameter is the parameter given when the procedure or function is declared, with an identifier in the heading. When the function or procedure is called, an actual parameter substitutes for the formal parameter given earlier; here the parameter takes the form of a variable or value or expression.

The following parameter features are available at the extended level:

1. A super array type can be passed as a reference parameter.

2. A reference parameter can be declared READONLY.

3. Explicit segmented reference parameters can be declared.

Value Parameters

When a value parameter is passed, the actual parameter is an expression. That expression is evaluated in the scope of the calling procedure or function and assigned to the formal parameter. The formal parameter is a variable local to the procedure or function called. Thus, formal value parameters are always local to a procedure or function.

Example of value parameters:

```
FUNCTION ADD (A, B, C : REAL): REAL;     {A, B, and C are
                                          formal parameters }

   .
   .
X := ADD (Y, ADD (1.111, 2.222, 3.333), (Z * 4) )
```

In the above function invocation, Y,ADD(1.111,2.222,3.333), and (Z * 4) are the expressions that make up the actual parameters. These expressions must all evaluate to the type REAL. The actual parameter expression must be assignment compatible with the type of the formal parameter.

Passing structured types by value is legal; however, it is inefficient, since the entire structure must be copied. A value parameter of a SET, LSTRING, or subrange type may also require a runtime error check if the $RANGECK is on. In addition, SET and LSTRING value parameters may require extra generated code for size adjustment.

A file variable or super array variable cannot be passed as a value parameter, since it cannot be assigned. However, a variable with a type derived from a super array or file buffer variable can be passed. Passing a file buffer variable as a value parameter implies normal evaluation of the buffer variable.

Reference Parameters

At the standard level, the keyword VAR precedes the formal parameter. Furthermore, the actual parameter must be a variable, not an expression. The formal parameter denotes this actual variable during the execution of the procedure. Any operation on the formal parameter is performed immediately on the actual parameter, by passing the machine address of the actual variable

to the procedure. This address is an offset into the default data segment.

Example of variable parameters:

```
PROCEDURE CHANGE_VARS (VAR A, B, C : INTEGER);{A, B, and C are
                                                formal reference
                                                parameters.}
    .
    .
    .
CHANGE_VARS (X, Y, Z);
```

In the above example, X, Y, and Z must be variables, not expressions. Also, the variables X, Y, and Z are altered whenever the formal parameters A, B, and C are altered in the declared procedure. This differs from the handling of value parameters, which can affect only the copies of values of variables.

If the selection of the variable involves indexing an array or dereferencing a pointer or address, these actions are executed before the procedure itself. The type of the actual parameter must be identical to the type of the formal parameter.

Passing a nonlocal variable as a VAR parameter puts a slash (/) or percent sign (%) in the G (global) column of the listing file. (See Chapter 14, Compiling, Linking, and Executing Programs, for information about significance of these characters in the G column of the listing). The following cannot be passed as VAR parameters:

1. a component of a PACKED structure (except CHAR of a STRING or LSTRING)

2. any variable with a READONLY attribute (includes CONST and CONSTS parameters and the FOR control variable)

Passing a file buffer variable by reference generates a warning message, because it bypasses the normal file system call generated by the use of any buffer variable. These calls are not generated when a file variable is passed by reference.

A VAR parameter passes an address that is really an offset into a default data segment. In some cases, access to objects residing in other segments is required. To pass these objects by reference, you must indicate to the compiler to use a segmented address containing both segment register and offset values. The extended level includes the parameter prefix VARS instead of VAR:

```
PROCEDURE CONCATS (VARS T, S:  STRING);
```

Note that a VARS can only be used as a data parameter in procedures and functions, not in the declaration section of

programs, procedures, and functions.

Super Array Parameters

Super array parameters may appear as formal reference parameters. This allows a procedure or function to operate on an array with a particular super array type (also a component type and index type), but without any fixed upper bounds. The formal parameter is a reference parameter of the super array type itself.

The actual parameter type must be a type derived from the super array type or the super array type itself (i.e., another reference parameter or dereferenced pointer). Except for comparing LSTRINGs, super array type parameters cannot be assigned or compared as a whole.

The actual upper and lower bounds of the array are available with the UPPER and LOWER functions; this permits routines that can operate on arrays of any size. An LSTRING actual parameter can be passed to a reference parameter of the super array type STRING. Therefore, the super array parameter STRING can be used for procedures and functions that operate on strings of both STRING and LSTRING types.

Example of super array parameters:

```
TYPE REALS = ARRAY [Ø..*] OF REAL;

PROCEDURE SUMRS (VAR X: REALS; CONST X: REALS);
BEGIN
  .
  .
END;
```

Constant and Segment Parameters

At the extended level, a formal parameter preceded by the reserved word CONST implies that the actual parameter is a READONLY reference parameter. This is especially useful for parameters of structured types, which may be constants, since it eliminates the need for a time-consuming value parameter copy. The actual parameter can be a variable, function result, or constant value.

No assignments can be made to the CONST parameter or any of its components. CONST super array types are permitted. A CONST parameter in one procedure cannot be passed as a VAR parameter to another procedure. However, it is permissible to pass a VAR parameter in one procedure as a CONST parameter in another.

Example of a CONST parameter:

```
PROCEDURE ERROR (CONST ERRMSG: STRING);
```

A CONST parameter passes an address that is really an offset into a default data segment. In some cases, access to objects residing in other segments is required. To pass these objects by reference, you must indicate to the compiler to use a segmented address that contains both segment register and offset values. The extended level includes the parameter prefix CONSTS, instead of CONST. Use of CONSTS parameters parallels use of VARS for formal reference parameters.

Example of a CONSTS parameter:

```
PROCEDURE CAT (VARS T: STRING;  CONSTS S: STRING);
```

Note that a CONSTS parameter can only be used as a data parameter in procedures and functions, not in the declaration section of programs, procedures, and functions.

You can also pass the value of an expression as a CONST or CONSTS parameter. The expression is evaluated and assigned to a temporary (hidden) variable in the frame of the calling procedure or function. You should enclose such an expression in parentheses to force its evaluation.

A function identifier can be passed by reference as a VAR, VARS, CONST, or CONSTS parameter. The function's local variable is passed, so the call must occur in the function's body or in a procedure or function declared with the function.

The value returned by a function designator can also be passed, like any expression, as a CONST or CONSTS parameter. Like any expression passed by reference, the function designator should be enclosed in parentheses, as follows:

```
PROCEDURE WRITE_ANSWER (CONSTS A: INTEGER);
BEGIN
   WRITELN ('THE ANSWER IS ,' A)
END;

FUNCTION ANSWER: INTEGER;
BEGIN
   ANSWER := 42;
   WRITE_ANSWER (ANSWER);{Pass reference to local variable.}
END;

PROCEDURE HITCH_HIKE;
BEGIN
   WRITE_ANSWER ((ANSWER)){Call ANSWER, assign to temporary
                           variable, pass reference to
                           temporary variable.}
END;
```

Procedural and Functional Parameters

When a procedural or functional parameter is passed, the actual identifier is that for a procedure or function. The formal parameter is a procedure or function heading, including any attributes, preceded by the reserved word PROCEDURE or FUNCTION.

For example, examine these declarations:

```
TYPE DOOR  = (FRONT, BARN, CELL, DOG_HOUSE);
     SPEED  = (FAST, SLOW, NORMAL);
     DIRECTION = (OPEN, SHUT);

PROCEDURE OPEN_DOOR_WIDE
   (VAR A : DOOR;  B : SPEED; C : DIRECTION);
    .
    .
PROCEDURE SLAM_DOOR
   (VAR DR : DOOR; SP : SPEED; DIR : DIRECTION);
    .
    .
PROCEDURE LEAVE_AJAR
   (VAR DD : DOOR; SS : SPEED; DD : DIRECTION);
```

All of the procedures in the example have parameter lists of equal length. The types of the parameters are not only compatible, but also identical. The formal parameters need not be identically named.

A procedural or functional parameter can accept one of these procedures if the procedure or function is set up correctly, as shown:

```
FUNCTION DOOR_STATUS (PROCEDURE MOVE_DOOR
   (VAR X: DOOR; Y: SPEED; Z: DIRECTION);
   VAR XX: DOOR; YY: SPEED; ZZ: DIRECTION):INTEGER;
   {"PROCEDURE MOVE_DOOR" is the formal procedural}
   {parameter; next two lines are other formal}
   {parameters.}

BEGIN {door_status}
   DOOR_STATUS := 0;
   MOVE_DOOR(XX, YY, ZZ);
   {One of the three procedures declared}
   {previously is executed here.}

   IF  XX = BARN AND ZZ = SHUT
   THEN DOOR_STATUS := 1;

   IF  XX = CELL AND ZZ = OPEN
   THEN DOOR_STATUS := 2
```

```
      IF  XX = DOG_HOUSE AND ZZ = SHUT
      THEN DOOR_STATUS := 3
END;
```

Use of the procedural parameter MOVEDOOR might occur in program statements as follows:

```
IF DOOR_STATUS
   (SLAM_DOOR, CELL, FAST, SHUT) = Ø
THEN
   SOCIETY := SAFE;
IF DOOR_STATUS
   (OPEN_DOOR_WIDE, BARN, SLOW, OPEN) = Ø
THEN
   COWS_ARE_OUT := TRUE;
IF DOOR_STATUS
   (LEAVE_AJAR, DOG_HOUSE, SLOW, OPEN) = Ø
THEN
   DOG_CAN_GET_IN := TRUE;
```

In each case above, the actual procedure list is compatible with the formal list, both in number and in type of parameters. If the parameter passed were a functional parameter, then the function return value would also have to be of an identical type.

In addition, the set of attributes for both the formal and actual procedural type must be the same, except that the PUBLIC and ORIGIN attributes and EXTERN directive are ignored.

A PUBLIC or EXTERN procedure, or any local procedure at any nesting level, can be passed to the same type of formal parameter. However, the PURE attribute and any calling sequence attributes must match. Also, in systems with segmented code addresses, a procedure or function passed as a parameter to an EXTERN procedure or function must itself be PUBLIC or EXTERN.

You cannot pass predeclared procedures and functions compiled as inline code; you can only pass them in called subroutines. Also, the READ, WRITE, ENCODE, and DECODE families are translated into other calls by the compiler, based on the argument types, and so cannot be passed. Corresponding routines in the file unit or encode/decode unit can be passed, however. For example, a READ of an INTEGER becomes a call to RTIFQQ and this procedure can be passed as a parameter.

The following intrinsic procedures and functions cannot be passed as procedure or function parameters:

1. at the standard level :

ABS	EOLN	PACK	SQR
ARCTAN	EXP	PAGE	SQRT
CHR	LN	PRED	SUCC
COS	NEW	READ	UNPACK

DISPOSE	ODD	READLN	WRITE
EOF	ORD	SIN	WRITELN

2. at the extended and system levels:

BYLONG	FLOAT4	READFN	SIZEOF
BYWORD	HIBYTE	READSET	TRUNC
DECODE	HIWORD	RESULT	TRUNC4
ENCODE	LOBYTE	RETYPE	UPPER
EVAL	LOWER	ROUND	WRD
FLOAT	LOWORD	ROUND4	

When a procedure or function passed as a parameter is finally activated, any nonlocal variables accessed are those in effect at the time the procedure or function is passed as a parameter, rather than those in effect when it is activated. Internally, both the address of the routine and the address of the upper frame (in the stack) are passed.

Example of formal procedure use:

```
PROCEDURE ALPHA;
  VAR I: INTEGER;

  PROCEDURE DELTA;
    BEGIN
      WRITELN('Delta done')
    END;

  PROCEDURE BETA (PROCEDURE XPR);
    VAR GLOB: INTEGER;

    PROCEDURE GAMMA;
      BEGIN GLOB := GLOB + 1 END;

    BEGIN  {Start BETA}
      GLOB := 0;
      IF I = 0
        THEN BEGIN
           I := 1;  XPR; BETA (GAMMA)
           END
        ELSE BEGIN
           GLOB := GLOB + 1; XPR
           END
    END;

  BEGIN  {Start ALPHA}
    I := 0;
    BETA (DELTA)
  END;
```

The following list describes the events that take place in the above example:

1. ALPHA is called.

2. BETA is called, passing the procedure DELTA.

3. This latter call creates an instance of GLOB on the stack (call it GLOB1).

4. BETA first clears GLOB1 by setting it to zero. Then, since I is Ø, the THEN clause is executed, which sets I to one and executes XPR, which is bound to DELTA.

5. Therefore, 'Delta done' is written to OUTPUT.

6. Now BETA is called recursively. BETA is passed GAMMA, and, at this time, the access path to any nonlocal variables used by GAMMA (i.e., GLOB1) is passed as well.

7. The second call to BETA creates another instance of GLOB (GLOB2). When GLOB2 is cleared this time, I is 1, so GLOB2 is incremented.

8. Then XPR is called, which is bound to GAMMA, so GAMMA is executed and increments the instance of GLOB active when GAMMA was passed to BETA, GLOB1.

9. GAMMA returns, the second BETA call returns, the first BETA call returns, and finally, ALPHA returns.

CHAPTER 11
AVAILABLE PROCEDURES AND FUNCTIONS

CONTENTS

Procedure ABORT

Function BYLONG

Function BYWORD

Function DECODE

Function ENCODE

Procedure EVAL

Function HIBYTE

Function HIWORD

Function LOBYTE

Function LOWER

Function LOWORD

Function RESULT

Function SIZEOF

Function UPPER

SYSTEM LEVEL INTRINSICS

Procedure FILLSC

Procedure MOVEL

Procedure MOVER

Procedure MOVESL

Procedure MOVESR

Function RETYPE

STRING INTRINSICS

Procedure CONCAT

Procedure COPYLST

Procedure COPYSTR

Procedure DELETE

Standard procedures and functions are "predeclared" in Pascal. This means that they do not have to be declared in a program and that they can be redefined. Pascal provides additional predeclared procedures and functions that are only available at the extended and system levels. They should be avoided if portability is necessary. Pascal also includes some useful library procedures and functions that you must declare EXTERN in order to use.

Pascal implements three kinds of procedures and functions:

1. Those that are predeclared, and the compiler translates them into other calls or special generated code (these you cannot pass as parameters).

2. Those that are predeclared but you call them normally (except for a name change).

3. Those that are not predeclared but available as part of the Pascal runtime library (these you must declare explicitly).

Procedures and functions are grouped according to implementation levels and functions. These groups are listed below:

Category	**Purpose**
File system	Operate on files of different modes and structures
Dynamic allocation	Dynamically allocate deallocate data structures on the heap at runtime
Data conversion	Convert data from one type to another
Arithmetic	Perform common transcendental and other numeric functions
Extended level intrinsics	Provide additional procedures and functions at the extended level of Pascal
System level intrinsics	Provide additional procedures and functions at the system level of Pascal
String intrinsics	Operate on STRING and LSTRING type data
Library	Available in the Pascal runtime library: they are not predeclared; you must declare them with the EXTERN directive

The File System procedures and functions are discussed separately in Chapter 12, File Oriented Procedures and Functions.

DYNAMIC ALLOCATION PROCEDURES

The procedures, NEW and DISPOSE, allow dynamic allocation and deallocation of data structures at runtime. NEW allocates a variable in the heap, and DISPOSE releases it.

Procedure DISPOSE (VARS P: POINTER); (Short Form)

This procedure releases the memory used for the variable pointed to by P. P must be a valid pointer; it may not be NIL, uninitialized, or pointing at a heap item that already has been DISPOSEd. These are checked if the NIL checking switch is on.

P should not be a reference parameter or a WITH statement record pointer, but these errors are not caught. A DISPOSE of a WITH statement record can be done at the end of the WITH statement without problem.

If the variable is a super array type or a record with variants, you may safely use the short form of DISPOSE to release the variable, regardless of whether it was allocated with the long or short form of NEW. Using the short form of DISPOSE on a heap variable allocated with the long form of NEW is an ISO-defined error not caught in this Pascal.

Procedure DISPOSE (VARS P: POINTER; T1, T2,...TN; TAGS); (Long Form)

This procedure works the same way as the short form. However, the long form checks the size of the variable against the size implied by the tag field or array upper bound values T1, T2, ...Tn. These tag values should be the same as defined in the corresponding NEW procedure. See also the SIZEOF function, which uses the same array upper bounds or tag value parameters to return the number of bytes in a variable.

Procedure NEW (VARS P: POINTER); (Short Form)

This procedure allocates a new variable V on the heap and at the same time assigns a pointer to V to the pointer variable P (a VARS parameter). The type of V is determined by the pointer declaration of P. If V is a super array type, you should use the long form of the procedure. If V is a record type with variants, the

variants giving the largest possible size are assumed, permitting any variant to be assigned to P^.

Procedure NEW (VARS P: POINTER; T1, T2,...TN: TAGS); (Long Form)

This procedure allocates a variable with the variant specified by the tag field values T1 through Tn. The tag field values are listed in the order in which they are declared. Any trailing tag fields can be omitted.

If all tag field values are constant, Pascal allocates only the amount of space required on the heap, rounded up to a word boundary. The value of any omitted tag fields is assumed to be such that the maximum possible size is allocated.

If some tag fields are not constant values, the compiler uses one of two strategies:

1. It assumes that the first nonconstant tag field and all following tags have unknown values, and allocates the maximum size necessary.

2. It generates a special runtime call to a function that calculates the record size from the variable tag values available. This depends on the implementation. A similar procedure applies to DISPOSE and SIZEOF.

You should set all tag fields to their proper values after the call to NEW and never change them. The compiler does not do any of the following:

1. assign tag values

2. check that they are initialized correctly

3. check that their value is not changed during execution

In ISO Pascal, a variable created with the long form of NEW cannot be any of the following:

1. used as an expression operand

2. passed as a parameter

3. assigned a value

This Pascal does not catch these errors. Fields within the record can be used normally.

Assigning a larger record to a smaller one allocated with the long form of NEW would wipe out part of the heap. This condition is difficult to detect at compile time. Therefore, any assignment to a record in the heap that has variants uses the actual length of the record in the heap, rather than the maximum length.

However, an assignment to a field in an invalid variant may destroy part of another heap variable or the heap structure itself. This error is not caught, unless all tag values are explicit, the tag values are correct.

The extended level allows pointers to super arrays. The long form of NEW is used as described above, except that array upper bound values are given instead of tag values. All upper bounds must be given. Bounds can be constants or expressions; in any case, only the size required is allocated.

The entire array referenced by such a pointer cannot be assigned or compared, except that LSTRINGs can always be compared. The entire array can be passed as a reference parameter if the formal parameter is of the same super array type. Components of the array can be used normally.

DATA CONVERSION PROCEDURES AND FUNCTIONS

You should use the following procedures and functions to convert data from one type to another:

Function CHR (X: ORDINAL): CHAR;

This function converts any ordinal type to CHAR. The ASCII code for the result is ORD (X). This is an extension to the ISO Pascal, which requires X to be of type INTEGER. An error occurs if ORD (X) > 255 or ORD (X) < Ø. However, the error is caught only if the range checking switch is on.

Function FLOAT (X: INTEGER): REAL;

This function converts an INTEGER value to a REAL value. You normally don't need this function, since INTEGER-to-REAL is usually done automatically. However, because FLOAT is needed by the runtime package, it is included at the standard level.

Function FLOAT4 (X: INTEGER4): REAL;

This function converts an INTEGER4 value to a REAL value. This type conversion is also done automatically; however, it is possible that you might lose precision.

Function ODD (X: ORDINAL): INTEGER;

This function tests the ordinal value X to see whether it is odd. ODD is TRUE only if ORD (X) is odd; otherwise it is FALSE.

Function ORD (X: VALUE): INTEGER;

This function converts to INTEGER any value of one of the types shown below:

Type of X	Return value
INTEGER	X
WORD <= MAXINT	X
WORD > MAXINT	X - 2 * (MAXINT + 1) (i.e., same 16 bits as at start!)
CHAR	ASCII code for X
Enumerated	Position of X in the type definition, starting with Ø
INTEGER4	Lower 16 bits (i.e., same as ORD(LOWORD(INTEGER4))
Pointer	Integer value of pointer

Procedure PACK (CONST A: UNPACKED; I: INDEX; VAR Z: PACKED);

This procedure moves elements of an unpacked array to a packed array. If A is an ARRAY [M..N] OF T and Z is a PACKED ARRAY [U..V] OF T, then PACK (A, I, Z) is the same as:

```
FOR J := U TO V DO Z [J] := A [J - U + I]
```

In both PACK and UNPACK, the parameter I is the initial index within A. The bounds of the arrays and the value of I must be reasonable; i.e., the number of components in the unpacked array A from I to M must be at least as great as the number of components in the packed array Z. The range checking switch controls checking of the bounds.

Function PRED (X: ORDINAL): ORDINAL;

This function determines the ordinal "predecessor" to X. The ORD of the result returned is equal to ORD (X) - 1. An error occurs if the predecessor is out of range or overflow occurs. These errors are caught if appropriate debug switches are on.

Function ROUND (X: REAL): INTEGER;

This function rounds X away from zero. X is of type REAL4 or REAL8; the return value is of type INTEGER.

Examples:

```
ROUND (1.6) is 2
        ROUND (-1.6) is -2
```

An error occurs if ABS (X + Ø.5) >= MAXINT.

Function ROUND4 (X: REAL): INTEGER4;

This function rounds real X away from zero. X is of type REAL4 or REAL8; the return value is of type INTEGER4.

Examples:

```
ROUND4 (1.6) is 2
        ROUND4 (-1.6) is -2
```

An error occurs if ABS (X + Ø.5) >= MAXINT4.

Function SUCC (X: ORDINAL): ORDINAL;

This function determines the ordinal "successor" to X. The ORD of the returned result is equal to ORD (X) + 1. An error occurs if the successor is out of range or overflow occurs. These errors are caught if appropriate debug switches are on.

Function TRUNC (X: REAL): INTEGER;

This function truncates X toward zero. X is of type REAL4 or REAL8, and the return value is of type INTEGER.

Examples:

```
TRUNC (1.6) is 1
```

TRUNC (-1.6) is -1

An error occurs if ABS (X - 1.Ø) >= MAXINT.

Function TRUNC4 (X: REAL): INTEGER4;

This function truncates real X towards zero. X is of type REAL4 or REAL8, and the return value is of type INTEGER4.

Examples:

TRUNC4 (1.6) is 1 TRUNC4 (-1.6) is -1

An error occurs if ABS (X - 1.Ø) >= MAXINT4.

Procedure UNPACK (CONSTS Z: PACKED; VARS A: UNPACKED; I: INDEX);

This procedure moves elements from packed array to an unpacked array. If A is an ARRAY [M..N] OF T, and Z is a PACKED ARRAY [U..V] OF T then the above call is the same as:

```
FOR J := U TO V DO A [J - U + I] := Z [J]
```

In both PACK and UNPACK, the parameter I is the initial index within A. The bounds of the arrays and the value of I must be reasonable; i.e., the number of components in the unpacked array A from I to M must be at least as great as the number of components in the packed array Z. The range checking switch controls checking of the bounds.

See also PROCEDURE PACK.

Function WRD (X: VALUE): WORD;

This function converts to WORD any of the types shown below:

Type of X	Return Value
WORD	X
INTEGER >= Ø	X
INTEGER < Ø	X + MAXWORD + 1 (i.e., same 16 bits as at start!)
CHAR	ASCII code for X

Enumerated	Position of X in the type definition, starting with Ø
INTEGER4	Lower 16 bits (i.e., same as LOWORD(INTEGER4)
Pointer	Word value of pointer

ARITHMETIC FUNCTIONS

All arithmetic functions take a CONSTS parameter of type REAL4 or REAL8, or a type compatible with INTEGER (labeled "numeric"). ABS and SQR also take WORD and INTEGER4 values.

All functions on REAL data types check for an invalid (uninitialized) value. They also check for particular error conditions and generate a runtime error message if an error condition is found.

If the math checking switch is on, errors in the use of the functions ABS and SQR on INTEGER, WORD, and INTEGER4 data generate a runtime error message. If the switch is off, the result of an error is undefined.

FUNCTION ABS (X: NUMERIC): NUMERIC;

This function returns the absolute value of X. Both X and the return value are of the same numeric type: REAL4, REAL8, INTEGER, WORD, or INTEGER4. Since WORD values are unsigned, ABS (X) always returns X if X is of type WORD.

FUNCTION ARCTAN (X: REAL): REAL;

This function returns the arc tangent of X in radians. Both X and the return value are of type REAL. To force a particular precision, you must declare ATSRQQ (CONSTS REAL4) and/or ATDRQQ (CONSTS REAL8) and use them instead.

```
FUNCTION ATSRQQ (CONSTS A: REAL4): REAL4;
FUNCTION ATDRQQ (CONSTS A: REAL8): REAL8;
```

FUNCTION COS (X: NUMERIC): REAL;

This function returns the cosine of X in radians. Both X and the return value are of type REAL. To force a particular precision, you must declare CNSRQQ (CONSTS REAL4) and/or CNDRQQ (CONSTS REAL8) and use them instead.

```
FUNCTION CNSRQQ (CONSTS A: REAL4): REAL4;
FUNCTION CNDRQQ (CONSTS A: REAL8): REAL8;
```

FUNCTION EXP (X: NUMERIC): REAL;

This function returns the exponential value of X (i.e., e to the X). Both X and the return value are of type REAL. To force a particular precision, you must declare EXSRQQ (CONSTS REAL4) and/or EXDRQQ (CONSTS REAL8) and use them instead.

```
FUNCTION EXSRQQ (CONSTS A: REAL4): REAL4;
FUNCTION EXDRQQ (CONSTS A: REAL8): REAL8;
```

FUNCTION LN (X: REAL): REAL;

This function returns the logarithm, base e, of X. Both X and the return value are of type REAL. To force a particular precision, you must declare LNSRQQ (CONSTS REAL4) and/or LNDRQQ (CONSTS REAL8) and use them instead. An error occurs if X is less than or equal to zero.

```
FUNCTION LNSRQQ (CONSTS A: REAL4): REAL4;
FUNCTION LNDRQQ (CONSTS A: REAL8): REAL8;
```

FUNCTION SIN (X: NUMERIC): REAL;

This function returns the sine of X in radians. Both X and the return value are of type REAL. To force a particular precision, you must declare SNSRQQ (CONSTS REAL4) and/or SNDRQQ (CONSTS REAL8) and use them instead.

```
FUNCTION SNSRQQ (CONSTS A: REAL4): REAL4;
FUNCTION SNDRQQ (CONSTS A: REAL8): REAL8;
```

FUNCTION SQR (X: NUMERIC): NUMERIC;

This function returns the square of X, where X is of type REAL, INTEGER, WORD, or INTEGER4.

FUNCTION SQRT (X): REAL;

This function returns the square root of X, where X is of type REAL. To force a particular precision, you must declare SRSRQQ (CONSTS REAL4) and/or SRDRQQ (CONSTS REAL8) and use them instead. An error occurs if X is less than Ø.

```
FUNCTION SRSRQQ (CONSTS A: REAL4): REAL4;
FUNCTION SRDRQQ (CONSTS A: REAL8): REAL8;
```

REAL Functions

The Pascal runtime library provides several additional REAL4 and REAL8 functions. If you use them, all variable parameters must be passed as VARS and the functions must be declared with the EXTERN directive.

```
FUNCTION ACSRQQ (CONSTS A: REAL4): REAL4;
FUNCTION ACDRQQ (CONSTS A: REAL8): REAL8;
```

These functions return the arc cosine of A. Both A and the return value are of type REAL4 or REAL8.

```
FUNCTION AISRQQ (CONSTS A: REAL4): REAL4;
FUNCTION AIDRQQ (CONSTS A: REAL8): REAL8;
```

These functions return the integral part of A, truncated toward zero. Both A and the return value are of type REAL4 or REAL8.

```
FUNCTION ANSRQQ (CONSTS A: REAL4): REAL4;
FUNCTION ANDRQQ (CONSTS A: REAL8): REAL8;
```

Like AISRQQ and AIDRQQ, these functions return the truncated integral part of A, but round away from zero. Both A and the return value are of type REAL4 or REAL8.

```
FUNCTION ASSRQQ (CONSTS A: REAL4): REAL4;
FUNCTION ASDRQQ (CONSTS A: REAL8): REAL8;
```

These functions return the arc sine of A. Both A and the return value are of type REAL4 or REAL8.

```
FUNCTION A2SRQQ (A, B: REAL4): REAL4;
FUNCTION A2DRQQ (A, B: REAL8): REAL8;
```

These functions return the arc tangent of (A/B). Both A and B, as well as the return value, are of type REAL4 or REAL8.

```
FUNCTION CHSRQQ (CONSTS A: REAL4): REAL4;
FUNCTION CHDRQQ (CONSTS A: REAL8): REAL8;
```

These functions return the hyperbolic cosine of A. Both A and the return value are of type REAL4 or REAL8.

```
FUNCTION LDSRQQ (CONSTS A: REAL4): REAL4;
FUNCTION LDDRQQ (CONSTS A: REAL8): REAL8;
```

These functions return the logarithm, base 10, of A. Both A and the return value are of type REAL4 or REAL8.

```
FUNCTION MDSRQQ (CONSTS A, B: REAL4): REAL4;
FUNCTION MDDRQQ (CONSTS A, B: REAL8): REAL8;
```

A modulo B, defined as:

MDSRQQ (A, B) = A - AISRQQ (A/B) * B

MDDRQQ (A, B) = A - AIDRQQ (A/B) * B

Both A and B are of type REAL4 or REAL8.

```
FUNCTION MNSRQQ (CONSTS A, B: REAL4): REAL4;
FUNCTION MNDRQQ (CONSTS A, B: REAL8): REAL8;
```

These functions return the value of A or B, whichever is smaller. Both A and B are of type REAL4 or REAL8.

```
FUNCTION MXSRQQ (CONSTS A, B: REAL4): REAL4;
FUNCTION MXDRQQ (CONSTS A, B: REAL8): REAL8;
```

These functions return the value of A or B, whichever is larger. Both A and B are of type REAL4 or REAL8.

```
FUNCTION PIDRQQ (CONSTS A: REAL8; CONSTS B: INTEGER4): REAL8;
FUNCTION PISRQQ (CONSTS A: REAL4; CONSTS B: INTEGER4): REAL4;
```

These functions return the value is A**B (A to the INTEGER power of B). A is of type REAL4 or REAL8.

```
FUNCTION PRSRQQ (A, B: REAL4): REAL4;
FUNCTION PRDRQQ (A, B: REAL8): REAL8;
```

These functions return the value A**B (A to the REAL power of B). Both A and B are of type REAL4 or REAL8.

```
FUNCTION SHSRQQ (CONSTS A: REAL4): REAL4;
FUNCTION SHDRQQ (CONSTS A: REAL8): REAL8;
```

These functions return the hyperbolic sine of A. A is of type REAL4 or REAL8.

```
FUNCTION THSRQQ (CONSTS A: REAL4): REAL4;
FUNCTION THDRQQ (CONSTS A: REAL8): REAL8;
```

These functions return the hyperbolic tangent of A. Both A and the return value are of type REAL4 or REAL8.

```
FUNCTION TNSRQQ (CONSTS A: REAL4): REAL4;
FUNCTION TNDRQQ (CONSTS A: REAL8): REAL8;
```

These functions return the tangent of A. Both A and the return value are of type REAL4 or REAL8.

Some common mathematical functions are not standard in Pascal, but are relatively simple to accomplish with program statements or to define as functions in a program. Some typical definitions are as follows:

```
SIGN (X)       is    ORD (X > Ø) - ORD (X < Ø)
POWER (X, Y)   is    EXP (Y * LN (X))
```

You could also write your own functions to do the same thing. For example:

```
FUNCTION POWER (A, B: REAL): REAL [PURE];
BEGIN
  IF A <= Ø
    THEN
      ABORT ('Nonplus real to power', 24, Ø);
  POWER := EXP (B * LN (A));
END;
```

EXTENDED LEVEL INTRINSICS

The following intrinsic procedures and functions are available at the extended level:

Procedure ABORT (CONST STRING, WORD, WORD);

This procedure halts program execution in the same way as an internal runtime error. The STRING (or LSTRING) is an error message. The string parameter is a CONST, not a CONSTS parameter. The first WORD is an error code (see Appendix D, "Error Messages," for error code allocations); the second WORD can be anything. The second WORD is sometimes used to return a file error

status code from the operating system.

The parameters, as well as any information about the machine state (program counter, frame pointer, stack pointer) and the source position of the ABORT call (if the $LINE and/or $ENTRY debugging switches are on), are given to you in a termination message or are available to the debugging package.

If the $RUNTIME switch is on, then error messages give the location of the procedure or function that has called the routine in which ABORT was called. If $RUNTIME is on, $LINE and $ENTRY should be off, and routines in a source file should only call other $RUNTIME routines.

Function BYLONG (INTEGER-WORD, INTEGER-WORD): INTEGER4;

This function converts WORDS or INTEGERs (or the LOWORDs of INTEGER4s) to an INTEGER4 value. BYLONG concatenates its operands:

```
BYLONG (A, B) =

ORD (LOWORD (A)) * 65535 + WRD (HIWORD (B))
```

If the first value is of type WORD, its most significant bit becomes the sign of the result.

Function BYWORD (ONE-BYTE, ONE-BYTE): WORD;

This function converts bytes (or the LOBYTEs of INTEGERs or WORDs) to a WORD value. It takes two parameters of any ordinal type. BYWORD returns a WORD with the first byte in the most significant part and the second byte in the least significant part:

```
BYWORD (A, B) = LOBYTE(A) * 256 + LOBYTE(B)
```

If the first value is of type WORD, its most significant bit becomes the sign of the result.

Function DECODE (CONST LSTR: LSTRING X: M: N): BOOLEAN;

This function converts the character string in the LSTRING to its internal representation and assigns this to X. If the character string is not a valid external ASCII representation of a value whose type is assignment compatible with X, DECODE returns FALSE and the value of X is undefined. When X is a subrange, DECODE returns FALSE if the value is out of range (regardless of the

setting of the range checking switch). Leading and trailing spaces and tabs in the LSTRING are ignored. All other characters in the LSTRING must be part of the representation.

X must be one of the types INTEGER, WORD, enumerated, one of their subranges, BOOLEAN, REAL4, REAL8, INTEGER4, or a pointer (address types need the .R or .S suffix). The LSTR parameter must reside in the default data segment.

Function ENCODE (VAR LSTR: LSTRING, X: M: N): BOOLEAN;

This function converts the expression X to its external ASCII representation and puts this character string into LSTR. Returns TRUE, unless the LSTRING is too small to hold the string generated. In this case, ENCODE returns FALSE and the value of the LSTR is undefined. ENCODE works exactly the same as the WRITE procedure, including the use of M and N parameters (see Chapter 12, File Oriented Procedures and Functions for a discussion of these parameters).

X must be one of the types INTEGER, WORD, enumerated, one of their subranges, BOOLEAN, REAL4, REAL8, INTEGER4, or a pointer (address types need the .R or .S suffix). The LSTR parameter must reside in the default data segment.

Procedure EVAL (EXPRESSION, EXPRESSION,...);

This procedure evaluates expression parameters only, but accepts any number of parameters of any type. EVAL is used to evaluate an expression as a statement; it is commonly used to evaluate a function for its side effects only, without using the function return value.

Function HIBYTE (INTEGER-WORD): BYTE;

This function returns the most significant byte of an INTEGER or WORD. The most significant byte may be the first or the second addressed byte of the word.

Function HIWORD (INTEGER4): WORD;

This function returns the high-order word of the four bytes of the INTEGER4. The sign bit of the INTEGER4 becomes the most significant bit of the WORD.

Function LOBYTE (INTEGER-WORD): BYTE;

This function returns the least significant byte of an INTEGER or WORD. The least significant byte may be the first or the second addressed byte of the word.

Function LOWER (EXPRESSION): VALUE;

This function takes a single parameter of one of the following types: array, set, enumerated, or subrange. The value returned by LOWER is one of the following:

1. the lower bound of an array
2. the first allowable element of a set
3. the first value of an enumerated type
4. the lower bound of a subrange

LOWER uses the type, not the value, of the expression. The value returned by LOWER is always a constant.

Function LOWORD (INTEGER4): WORD;

This function returns the low-order WORD of the four bytes of the INTEGER4.

Function RESULT (FUNCTION-IDENTIFIER): VALUE;

This function is used to access the current value of a function. It can only be used within the body of the function itself or in a procedure or function nested within it.

Function SIZEOF (VARIABLE): WORD;
Function SIZEOF (VARIABLE, TAG1, TAG2,... TAGN): WORD;

This function returns the size of a variable in bytes. Tag values or array upper bounds are set as in the NEW and DISPOSE functions. If the variable is a record with variants, and the first form is used, the maximum size possible is returned. If the variable is a super array, the second form, which gives upper bounds, must be used.

Function UPPER (EXPRESSION): VALUE;

This function takes a single parameter of one of the

following types: array, set, enumerated, or subrange. The value returned by UPPER is one of the following:

1. the upper bound of an array

2. the last allowable element of a set

3. the last value of an enumerated type

4. the upper bound of a subrange

The value returned by UPPER is always a constant, unless the expression is of a super array type. In this case, the actual upper bound of the super array type is returned. Note that the type and not the value of the expression is used for UPPER.

SYSTEM LEVEL INTRINSICS

The system intrinsic feature provides the following procedures and functions:

Procedure FILLC (D: ADRMEM; N: WORD; C: CHAR);

This procedure fills D with N copies of the CHAR C. No bounds checking is done. The MOVE and FILL procedures take value parameters of type ADRMEM and ADSMEM, but since all ADR (or ADS) types are compatible, the ADR (or ADS) of any variable or constant can be used as the actual parameter. These are dangerous but sometimes useful procedures.

Procedure FILLSC (D: ADSMEM; N: WORD; C: CHAR);

This procedure fills D with N copies of the CHAR C. No bounds checking is done. The MOVE and FILL procedures take value parameters of type ADRMEM and ADSMEM, but since all ADR (or ADS) types are compatible, the ADR (or ADS) of any variable or constant can be used as the actual parameter. These are dangerous but sometimes useful procedures.

Procedure MOVEL (S, D: ADRMEM; N: WORD);

This procedure moves N characters (bytes) starting at S^ to D^, beginning with the lowest addressed byte of each array. Regardless of the value of the range and index checking switches, there is no bounds checking.

Example:

```
MOVEL (ADR 'New String Value', ADR V, 16)
```

You must use MOVEL and MOVESL to shift bytes left or when the address ranges do not overlap. The MOVE and FILL procedures take value parameters of type ADRMEM and ADSMEM, but since all ADR (or ADS) types are compatible, the ADR (or ADS) of any variable or constant can be used as the actual parameter. These are dangerous but sometimes useful procedures.

Procedure MOVER (S, D: ADRMEM; N: WORD);

This procedure is like MOVEL, but starts at the highest addressed byte of each array. Use MOVER and MOVESR to shift bytes right. As with MOVEL, there is no bounds checking.

Example:

```
MOVER (ADR V[Ø], ADR V[4], 12)
```

The MOVEs and FILLs take value parameters of type ADRMEM and ADSMEM, but since all ADR (or ADS) types are compatible, the ADR (or ADS) of any variable or constant can be used as the actual parameter. These are dangerous but sometimes useful procedures.

Procedure MOVESL (S, D: ADSMEM; N: WORD);

This moves N characters (bytes) starting at S^ to D^, beginning with the lowest addressed byte of each array. Regardless of the value of the range and index checking switches, there is no bounds checking.

Example:

```
MOVESL (ADS 'New String Value', ADS V, 16)
```

You must use MOVEL and MOVESL to shift bytes left or when the address ranges do not overlap. The MOVE and FILL procedures take value parameters of type ADRMEM and ADSMEM, but since all ADR (or ADS) types are compatible, the ADR (or ADS) of any variable or constant can be used as the actual parameter. These are dangerous but sometimes useful procedures.

Procedure MOVESR (S, D: ADSMEM; N: WORD);

This procedure is like MOVESL, but starts at the highest addressed byte of each array. Use MOVER and MOVESR to

shift bytes right. As with MOVESL, there is no bounds checking.

Example:

```
MOVER (ADR V[Ø], ADR V[4], 12)
```

The MOVE and FILL procedures take value parameters of type ADRMEM and ADSMEM, but since all ADR (or ADS) types are compatible, the ADR (or ADS) of any variable or constant can be used as the actual parameter. These are dangerous but sometimes useful procedures.

Function RETYPE (TYPE-IDENT, EXPRESSION): TYPE-IDENT;

This function provides a generic type escape, returns the value of the given expression as if it had the type named by the type identifier. The types implied by the type identifier and the expression should usually have the same length, but this is not required. RETYPE for a structure can be followed by component selectors (array index, fields, reference, etc.). RETYPE is a "dangerous" type escape and may not work as intended.

Example:

```
TYPE COLOR = (RED, BLUE, GREEN);
     S2    = STRING (2);

 VAR C :#CHAR;
     I, J :#INTEGER;
     R :#REAL4;     TINT:#COLOR;
     .
     .
     R := RETYPE (REAL4, 'abcd');

     {{Here, a 4-byte string literal is converted into a
     real number note that REAL4 numbers also require 4
     bytes.}

     TINT := RETYPE (COLOR, 2)

     { {Here, 2 is converted into a color which in this
     case is GREEN.  This, is a relatively "safe" use of
     the RETYPE function.}


     C := RETYPE (S2, I) [J]

     {Here, I is retyped into a two character string.
     Then J selects a single character of the string
     which is assigned to C.}
```

There are two other ways to change type in Pascal:

1. First, you can declare a record with one variant of each type needed, assign an expression to one variant, and then get the value back from another variant. (This is an error not caught at the standard level.)

2. Second, you can declare an address variable of the type wanted and assign to it the address of any other variable (using ADR).

   Each of these methods has its own subtle differences and quirks and should be avoided whenever possible.

STRING INTRINSICS

The string intrinsics feature provides a set of procedures and functions, some of which operate on STRINGs and LSTRINGs, and some on LSTRINGs only.

Procedure CONCAT (VARS D: LSTRING; CONSTS S: STRING);

This procedure concatenates S to the end of D. The length of D increases by the length of S. An error occurs if D is too small, i.e., if UPPER (D) < D.LEN + UPPER (S).

Procedure COPYLST (CONSTS S: STRING; VARS D: LSTRING);

This procedure copies S to LSTRING D. The length of D is set to UPPER (S). An error occurs if the length of S is greater than the maximum length of D, i.e., if UPPER (S) > UPPER (D).

Procedure COPYSTR (CONSTS S: STRING; VAR D: STRING);

This procedure copies S to STRING D. The remainder of D is set to blanks if UPPER (S) < UPPER (D). An error occurs if the length of S is greater than the maximum length of D, i.e., if UPPER (S) > UPPER (D).

Procedure DELETE (VARS D: LSTRING; I, N: INTEGER);

This procedure deletes N characters from D, starting with D [I]. An error occurs if an attempt is made to delete more characters starting at I than it is possible to delete, i.e., if D.LEN < (I + N - 1).

Procedure INSERT (CONSTS S: STRING; VARS D: LSTRING; I: INTEGER):

This procedure inserts S starting just before D [I]. An error occurs if D is too small, i.e., if

UPPER (D) < UPPER (S) + D.LEN + 1

or if:

D.LEN < I

Procedure POSITN (CONSTS PAT: STRING; CONSTS S: STRING; I: INTEGER): INTEGER

This function returns the integer position of the pattern PAT in S, starting the search at S [I]. If PAT is not found or if I > upper (S), the return value is Ø. If PAT is the null string, the return value is 1. There are no error conditions.

Function SCANEQ (LEN: INTEGER; PAT: CHAR; CONST S: STRING; I: INTEGER): INTEGER;

This function scans, starting at S [I], and returns the number of characters skipped. SCANEQ stops scanning when a character equal to pattern PAT is found or LEN characters have been skipped. If LEN < Ø, SCANEQ scans backwards and returns a negative number. SCANEQ returns the LEN parameter if it finds no characters equal to pattern PAT found or if I > UPPER (S). There are no error conditions.

Function SCANNE (LEN: INTEGER; PAT: CHAR; CONST S: STRING; I: INTEGER): INTEGER;

This function is like SCANEQ, but stops scanning when a character not equal to pattern PAT is found. Scans, starting at S [I], and returns the number of characters skipped. SCANEQ stops scanning when a character not equal to pattern PAT is found or LEN characters have been skipped. If LEN < Ø, SCANEQ scans backwards and returns a negative number. SCANEQ returns LEN parameter if it finds all characters equal to pattern PAT found or if I > UPPER (S). There are no error conditions.

LIBRARY PROCEDURES AND FUNCTIONS

The following routines are not predeclared, but are available to you in the Pascal runtime library. You must declare them, with the EXTERN directive, before using them in a program.

Initialization and Termination Routines

PROCEDURE BEGOQQ;

This procedure is called during initialization, and the default version does nothing. However, you may write your own version of BEGOQQ, if you want, to invoke a debugger or to write customized messages, such as the time of execution, to to a terminal screen.

PROCEDURE BEGXQQ;

After your program is linked and loaded, BEGXQQ is the defined entry point for the load module. As the overall initialization routine, BEGXQQ performs the following actions:

1. It resets the stack and the heap.
2. It initializes the file system.
3. It calls BEGOQQ.
4. It calls the program body.

Invoking this procedure to restart a program does not take care of closing any files that may have previously been opened. Similarly, it does not re-initialize variables originally set in a VALUE section or with the initialization switch on.

PROCEDURE ENDOQQ;

This procedure is called during termination and the default version does nothing. However, you may write your own version of ENDOQQ, if you want, to invoke a debugger or to write customized messages, such as the time of execution, to a terminal screen. Since ENDOQQ is called after errors are processed, if ENDOQQ itself invokes an error, the result is an infinite termination loop.

PROCEDURE ENDXQQ;

This procedure is the overall termination routine and performs the following actions:

1. It calls ENDOQQ.
2. It terminates the file system (closing any open files).
3. It returns to the operating system (or whatever called BEGXQQ).

   ENDXQQ may be useful for ending program execution from inside a procedure or function, without calling ABORT.

Heap Management

FUNCTION PreAllocHeap (VARS cbAlloc: WORD); ErcType;

This function allows the user to specify how much space they would like dedicated to the Pascal heap. The heap will grow to this amount and then stop. The user can use short lived memory without worrying about overlapping memory with the heap. CbAlloc is the count of bytes to allocate for the heap. If cbAlloc is #ØFFFF then the maximum storage will be allocated for the heap. ErcType is a BTOS error code. If the function is successful, the BTOS status will be Ø, otherwise an operating system error was detected.

NO-overflow Arithmetic Functions

These functions implement 16-bit and 32-bit modulo arithmetic. Overflow or carry is returned, instead of invoking a runtime error.

FUNCTION LADDOK (A, B: INTEGER4; VAR C: INTEGER4): BOOLEAN;

This function sets C equal to A plus B. It is one of two functions that do 32-bit signed arithmetic without causing a runtime error, even if the arithmetic debugging switch is on. Both LADDOK and LMULOK return TRUE if there is no overflow, and FALSE if there is. These routines are useful for extended-precision arithmetic, or modulo 2^32 arithmetic, or arithmetic based on user input data.

FUNCTION LMULOK (A, B: INTEGER4; VAR C: INTEGER4): BOOLEAN;

This function sets C equal to A times B. It is one of two functions that do 32-bit signed arithmetic without causing a runtime error on overflow. Normal arithmetic may cause a runtime error even if the arithmetic debugging switch is off. Both LMULOK and LADDOK return TRUE if there is no overflow, and FALSE if there is. These routines are useful for extended-precision arithmetic, or modulo 2^32 arithmetic, or arithmetic based on user input data.

FUNCTION SADDOK (A, B: INTEGER; VAR C: INTEGER): BOOLEAN;

This function sets C equal to A plus B. It is one of two functions that do 16-bit signed arithmetic without causing a runtime error on overflow. Normal arithmetic may cause a runtime error even if the arithmetic debugging switch is off. Both SADDOK and SMULOK return TRUE if there is no overflow, and FALSE if there is. These routines can be useful for extended-precision arithmetic, or modulo 2^16 arithmetic, or arithmetic based on user input data.

FUNCTION SMULOK (A, B: INTEGER; VAR C: INTEGER): BOOLEAN;

This function sets C equal to A times B. It is one of two functions that do 16-bit signed arithmetic without causing a runtime error on overflow. Normal arithmetic may cause a runtime error, even if the arithmetic debugging switch is off. Each routine returns TRUE if there is no overflow, and FALSE if there is. These routines can be useful for extended-precision arithmetic, or modulo 2^16 arithmetic, or arithmetic based on user input data.

FUNCTION UADDOK (A, B: WORD; VAR C: WORD): BOOLEAN;

This function sets C equal to A plus B. It is one of two functions that do 16-bit unsigned arithmetic without causing a runtime error on overflow. Normal arithmetic may cause a runtime error even if the arithmetic debugging switch is off. The following is the binary carry resulting from this addition of A and B:

WRD (NOT UADDOK (A, B, C)))

Both UADDOK and UMULOK return TRUE if there is no overflow and FALSE if there is. These routines are useful for extended-precision arithmetic, or modulo 2^16 arithmetic, or arithmetic based on user input data.

FUNCTION UMULOK (A, B: WORD; VAR C: WORD): BOOLEAN;

This function sets C equal to A times B. It is one of two functions that do 16-bit unsigned arithmetic without causing a runtime error on overflow. Normal arithmetic may cause a runtime error even if the arithmetic debugging switch is off. Each routine returns TRUE if there is no overflow and FALSE if there is. These routines are useful for extended-precision arithmetic, or modulo 2^16 arithmetic, or arithmetic based on user input data.

CHAPTER 12
FILE-ORIENTED PROCEDURES AND FUNCTIONS

CONTENTS

Chapter 11, "Available Procedures and Functions," described eight categories of procedures and functions that are available to you either because they are predeclared or because they are part of the Pascal runtime library. All except those that relate to file input and output were discussed in detail.

This Chapter discusses all of the file I/O procedures and functions, as well as lazy evaluation which is a special feature that facilitates your use of files.

The Pascal file system supports a variety of procedures and functions that operate on files of different modes and structures. These procedures and functions can be categorized as follows:

Category	Procedures	Functions
Primitive	GET	EOF
	PAGE	EOLN
	PUT	
	RESET	
	REWRITE	
Textfile I/O	READ	
	READLN	
	WRITE	
	WRITELN	
Extended Level I/O	ASSIGN	
	CLOSE	
	DISCARD	
	READSET	
	READFN	
	SEEK	

FILE SYSTEM PRIMITIVE PROCEDURES AND FUNCTIONS

This section describes the seven primitive file system procedures and functions, which perform file I/O at the most basic level. Later descriptions of READ and WRITE procedures are defined in terms of the primitives GET and PUT. In all descriptions which follow, F is a file parameter (files are always reference parameters), and F^ is the buffer variable.

All file variables operated on by these procedures must reside in the default data segment. This restriction increases the efficiency of file system calls.

EOF and EOLN

The functions EOF and EOLN check for end-of-file and end-of-line conditions, respectively. They return a BOOLEAN result. In general, these values indicate when to stop reading a line or a file.

FUNCTION EOF: BOOLEAN;
FUNCTION EOF (VAR F): BOOLEAN;

This function indicates whether the buffer variable F^ is positioned at the end of the file F for SEQUENTIAL and TERMINAL file modes. Therefore, if EOF (F) is TRUE, either the file is being written or the last GET has reached the end of the file.

With the DIRECT file mode, if EOF (F) is TRUE, either the last operation was a WRITE (the file may or may not be positioned at the end in this case) or the last GET reached the end of the file.

EOF without a parameter is equivalent to EOF (INPUT). EOF (INPUT) is generally never TRUE, except when INPUT is reassigned to another file. Calling the EOF (F) function accesses the buffer variable F^.

FUNCTION EOLN: BOOLEAN;
FUNCTION EOLN (VAR F): BOOLEAN;
This function indicates whether the current position of the file is at the end of a line in the textfile F after a GET (F). The file must have ASCII structure.

According to the ISO standard, calling EOLN (F) when EOF (F) is TRUE is an error. In this Pascal, this error is caught in most cases. The file F must be a file of type TEXT.

If EOLN (F) is TRUE, the value of F^ is a space, but the file is positioned at a line marker. EOLN without a parameter is equivalent to EOLN (INPUT). Calling the EOLN (F) function accesses the buffer variable F^.

GET and PUT

The primitive procedures GET and PUT are used to read to and write from the buffer variable, F^. GET assigns the next component of a file to the buffer variable. PUT performs the inverse operation and writes the value of the buffer variable to the next component of the file F.

PROCEDURE GET (VAR F);

If there is a next component in the file F, then:

1. The current file position is advanced to the next component.

2. The value of this component is assigned to the buffer variable F^.

3. EOF (F) becomes FALSE.

Advancing and assigning may be deferred internally, depending on the mode of the file.If no next component exists, then EOF (F) becomes TRUE and the value of F^ becomes undefined. EOF (F) must be FALSE before GET (F), since reading past the end of file produces a runtime error.

However, if F has mode DIRECT, EOF (F) can be TRUE or FALSE, since DIRECT mode permits repeated GET operations at the end of the file. If F^ is a record with variants, the compiler reads the variant with the maximum size.

PROCEDURE PUT (VAR F);

This procedure writes the value of the file buffer variable F^ at the current file position and then advances the position to the next component.

1. For SEQUENTIAL and TERMINAL mode files, PUT is permitted if the previous operation on F was a REWRITE, PUT, or other WRITE procedure, and if it was not a RESET, GET, or other READ procedure.

2. For DIRECT mode files, PUT may occur mmediately after a RESET or GET. Exceptions to these rules cause errors to be generated. The value of F^ always becomes undefined after a PUT.

EOF (F) must be TRUE before PUT (F), unless F is a DIRECT mode file. EOF (F) is always TRUE after PUT (F). If F^ is a record with variants, the variant with the maximum size is written.

RESET and REWRITE

The procedures RESET and REWRITE are used to set the current position of a file to its beginning. RESET is used to prepare for later GET and READ operations.

REWRITE is used to prepare for later PUT and WRITE operations.

PROCEDURE RESET (VAR F);

This procedure resets the current file position to its beginning and does a GET (F). If the file is not empty, the first component of F is assigned to the buffer variable F^, and EOF (F) becomes false. If the file is empty, the value of F^ is undefined and EOF (F) becomes true. RESET initializes a file F prior to its being read. For DIRECT files, writing can be done after RESET as well.

A RESET closes the file and then opens it in a way that is dependent on the operating system. An error occurs if the filename has not been set (as a program parameter or with ASSIGN or READFN) or if the file cannot be found by the operating system. If an error occurs during RESET, the file is closed, even if the file was opened correctly and the error came with the initial GET.

RESET (INPUT) is done automatically when a program is initialized, but is also allowed explicitly. RESET on a file with mode DIRECT allows either reading or writing, but the file is not created automatically. Also, the initial GET reads record number one on a DIRECT mode file.

Note that an explicit GET (F) immediately following a RESET (F) assigns the second component of the file to the buffer variable. However, a READ (F, X) following a RESET (F) sets X to the first component of F, since READ (F, X) is "X := F^; GET (F)".

PROCEDURE REWRITE (VAR F);

This procedure positions the current file to its beginning. The value of F^ is undefined and EOF (F) becomes TRUE. This is needed to initialize a file F before writing (for DIRECT files, reading can be done after REWRITE too).

A REWRITE closes the file and then opens it in a way that is dependent on the operating system. If the file does not exist in the operating system, it is created. If it does exist, its old value is lost (unless it has mode DIRECT). The filename must have been set (as a program parameter or with ASSIGN or READFN).

If an error occurs during REWRITE, the file is closed. An existing file with the same name is not affected when

a REWRITE error occurs.

REWRITE (OUTPUT) is done automatically when a program is initialized, but can also be done explicitly if desired. REWRITE on a DIRECT mode file allows both reading and writing. REWRITE does not do an initial PUT the way RESET does an initial GET.

PAGE

The procedure PAGE helps in formatting textfiles. It is not a "necessary" procedure in the same sense as GET and PUT.

PROCEDURE PAGE;
PROCEDURE PAGE (VAR F);

This procedure causes skipping to the top of a new page when the textfile F is printed. Since PAGE writes to the file, the initial conditions described for PUT must be TRUE. The file must have ASCII structure. PAGE without a parameter is equivalent to PAGE (OUTPUT).

If F is not positioned at the start of a line, PAGE (F) first writes a line marker to F. If F has mode SEQUENTIAL or DIRECT, then PAGE (F) writes a form feed, CHR (12). If F has mode TERMINAL, the effect is defined by the operating system interface, which will usually also write a form feed.

Lazy Evaluation

Lazy evaluation is designed to solve a recurring problem in Pascal, specifically, reading from a terminal in a natural way. The underlying problem is that the ISO standard defines the procedure RESET with an initial GET.

Although acceptable in Pascal's original batch processing, sequential file environment, this kind of read-ahead doesn't work for interactive I/O. Lazy evaluation provides for deferring actual physical input (textfiles only) when a buffer variable is evaluated.

For example, if a normal file is RESET and then READ, the RESET procedure calls the GET procedure, which sets the buffer variable to the first component of the file. However, if the file is a terminal, this first component does not yet exist!

Therefore, at a terminal, you must first type a character to accommodate the GET procedure. Only then would you be prompted for any input. Lazy evaluation eliminates this problem for textfiles by giving the file's buffer variable a special status

value that is either "full" or "empty."

The normal condition after a GET (F) is empty. The status is full after a buffer variable has been assigned to or assigned from. Full implies that the buffer variable value is equal to the currently pointed-to component. Empty implies just the opposite, that the buffer variable value does not equal the value of the currently pointed to component and input to the buffer variable has been deferred.

These rules are summarized as follows:

Statement	**Status at call**	**Action**	**Status on exit**
GET (F)	Full	Point to next file component. Becomes EMPTY since value pointed to is not in buffer variable.	Empty
GET (F)	Empty	Load buffer variable with current file component, then point to next file component. Becomes EMPTY since value pointed to is not in buffer variable.	Empty
Reference to F^	Full	No action required.	Full
Reference to F^	Empty	Load buffer variable with current file component.	Full

Note that RESET (F) first sets the status full and then calls GET, which sets the status to empty without any physical input.

Example of lazy evaluation with automatic REWRITE call:

```
{INPUT is automatically a textfile.}
{RESET (INPUT); done automatically.}
WRITE (OUTPUT, "Enter number: ");
READLN (INPUT, FOO);
```

The automatic initial call to the RESET procedure calls a GET procedure, which changes the buffer variable status from full to empty. The first physical action to the terminal is the prompt output from the WRITE. READLN does a series of the following operations:

```
temp := INPUT^;
GET (INPUT)
```

Physical input occurs when each INPUT^ is fetched and the GET procedure sets the status back to empty.

READLN ends with the sequence:

```
WHILE NOT EOLN DO GET (INPUT);
GET (INPUT)
```

This operation skips trailing characters and the line marker. The EOLN function invokes the physical input. Entering the carriage return sets the EOLN status. Both the GET procedure in the WHILE loop and the trailing GET set the status back to empty. The last physical input in the sequence above is reading the carriage return.

TEXT FILE INPUT AND OUTPUT

Human-readable input and output in standard Pascal are done with textfiles. Textfiles are files of type TEXT and always have ASCII structure. Normally, the standard textfiles INPUT and OUTPUT are given as program parameters in the PROGRAM heading:

```
PROGRAM IN_AND_OUT (INPUT,OUTPUT);
```

Other textfiles usually represent some input or output device such as a terminal, a card reader, a line printer, or an operating system disk file. The extended level permits using additional files not given as program parameters. In order to facilitate the handling of textfiles, the four standard procedures READ, READLN, WRITE, and WRITELN are provided in addition to the procedures GET and PUT.

These procedures are more flexible in the syntax for their parameter lists, allowing, among other things, for a variable number of parameters. Moreover, the parameters need not necessarily be of type CHAR, but can also be of certain other types, in which case the data transfer is accompanied by an implicit data conversion operation. In some cases, parameters can include additional formatting values that affect the data conversions used.

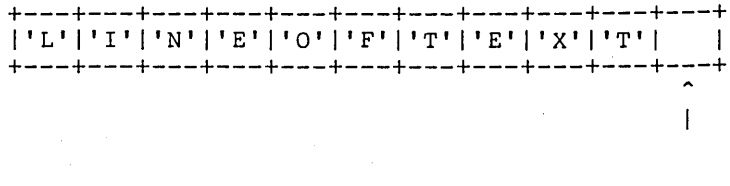
If the first variable is a file variable, then it is the file to be read or written. Otherwise, the standard files INPUT and OUTPUT are automatically assumed as default values in the cases of reading and writing, respectively.

These two files have TERMINAL mode and ASCII structure and are predeclared as:

```
VAR INPUT, OUTPUT:  TEXT;
```

The files INPUT and OUTPUT are treated like other textfiles. They can be used with ASSIGN, CLOSE, RESET, REWRITE, and the other procedures and functions. However, even if present as program parameters, they are not initialized with a filename. Instead, they are assigned to the user's terminal. RESET of INPUT and REWRITE of OUTPUT are done automatically, whether or not they are present as program parameters.

Textfiles represent a special case among file types insofar as they are structured into lines by "line markers". If, upon reading a textfile F, the file position is advanced to a line marker (i.e., past the last character of a line), then the value of the buffer variable F^ becomes a blank, and the standard function EOLN (F) yields the value true. For example:



Advancing the file position once more causes one of three things to happen:

1. If the end of the file is reached, then EOF (F) becomes TRUE.

2. If the next line is empty, a blank is assigned to F^ and EOLN (F) remains TRUE.

3. Otherwise, the first character of the next line is assigned to F^ and EOLN (F) is set to FALSE.

Since line markers are not elements of type CHAR in standard Pascal, they can, in theory, only be generated by the procedure WRITELN. However, in this Pascal, an actual character may be used for the line marker. It may therefore be possible to WRITE a line marker, but not to READ one.

When a textfile being written is closed, a final line marker is automatically appended to the last line of any nonempty file in which the last character is not already a line marker.

When a textfile being read reaches the end of a nonempty file, a line marker for the last line is returned even if one was not present in the file. Therefore, lines in a textfile always end with a line marker.

Any list of data written by a WRITELN is usually readable with the same list in a READLN (unless an LSTRING occurs that is not on the end of the list.)

Interactive prompt and response is very easy in Pascal. To have input on the same line as the response, use WRITE for the prompt. READLN must always be used for the response. For example:

```
WRITE ('Enter command:  ');
READLN (response);
```

If no file is given, most of the textfile procedures and functions assume either the INPUT file or the OUTPUT file. For example, if I is of type INTEGER, then READ (I) is the same as READ (INPUT, I).

READ and READLN

PROCEDURE READ
PROCEDURE READLN

READ and READLN read data from textfiles. Both are defined in terms of the more primitive operation, GET. That is, if P is of type CHAR, then READ (F, P) is equivalent to:

```
BEGIN
   P := F^; {Assign buffer variable F^ to P.}
   GET (F)  {Assign next component of file to F^.}
END
```

READ can take more than a single parameter, as in READ (F, P1, P2, ... Pn). This is equivalent to the following:

```
BEGIN
   READ (F, P1);
   READ (F, P2);
     .
     .
   READ (F, Pn)
END
```

The procedure READLN is very much like READ, except that it reads up to and including the end-of-line. At the primitive GET level, without parameters, READLN is equivalent to the following:

```
BEGIN
   WHILE NOT EOLN (F) DO GET (F);
   GET (F)
END
```

A READLN with parameters, as in READLN (F, P1, P2,... Pn), is equivalent to the following:

```
BEGIN
   READ (F, P1, P2, Pn);
   READLN (F)
END
```

READLN is often used to skip to the beginning of the next line. It can only be used with textfiles (ASCII mode).

If no other file is specified, both READ and READLN read from the standard INPUT file. Therefore, the name INPUT need not be designated explicitly. For example, these two READ statements perform identical actions:

```
READ (P1, P2, P3)
READ (INPUT, P1, P2, P3) {Reads INPUT by default}
```

At the standard level, parameters P1, P2, and P3 above must be of one of the following types:

```
CHAR
INTEGER
REAL
```

The extended level also allows READ variables of the following types:

```
WORD
an enumerated type
BOOLEAN
INTEGER4
a pointer type
STRING
LSTRING
```

When the compiler reads a variable of a subrange type, the value read must be in range. Otherwise, an error occurs, regardless of the setting of the range checking switch.

The procedure READ can also read from a file that is not a textfile (e.g., has BINARY mode). The form READ (F, P1, P2, ... Pn) can be used on a BINARY file. However, this READ will not work as expected after a SEEK on a DIRECT mode file. For BINARY files, READ (F, X) is equivalent to:

```
BEGIN
   X := F^;
   GET (F)
END
```

READ Formats

The READ process for formatted types (everything except CHAR, STRING, and LSTRING) first reads characters into an internal LSTRING and then decodes the string to get the value.

Two important points apply to formatted reads:

1. Leading spaces, tabs, form feeds, and line markers are skipped. For example, when doing READLN (I, J, K) where I, J, and K are integers, the numbers can all be on the same line or spread over several lines.

2. Characters are read as long as they are in the set of characters valid for the type wanted. For example, "-1-2-3" is read as the string of characters for a single INTEGER, but gives an error when the string is decoded. This means that items should be separated by spaces, tabs, line markers, or characters not permitted in the format reads.

Most of the formatting rules below apply to the function DECODE, as well.

1. INTEGER and WORD types

   If P is of type INTEGER, WORD, or a subrange thereof, then READ (F, P) implies reading a sequence of characters from F which form a number according to the normal Pascal syntax, and then assigning the number to P. Nondecimal notation (16#C007, 8#74, 10#19, 2#101, #Face) is accepted for both INTEGER and WORD, with a radix of 2 through 36. If P is of an INTEGER type, a leading plus (+) or minus (-) sign is accepted. If P is of a WORD type, then numbers up to MAXWORD are accepted (32768..65535).

2. REAL and INTEGER4 types

   If P is of type REAL, or at the extended level type INTEGER4, READ (F, P) implies reading a sequence of characters from F that form a number of the appropriate type and assigning the number to P. Nondecimal notation is not accepted for REAL numbers, but is accepted for INTEGER4 numbers. When reading a REAL value, a number with a leading or trailing decimal point is accepted, even though this form gives a warning if used as a constant in a program.

3. Enumerated and Boolean types

At the extended level, if P is an enumerated type or BOOLEAN, a number is read as a WORD subrange and a value assigned to P such that the number is the ORD of the enumerated type's value. In addition, if P is type BOOLEAN, reading one of the character sequences 'TRUE' or 'FALSE' cause true and false, respectively, to be assigned to P. The number read must be in the range of the ORD values of the variable.

4. Reference types

At the extended level, if P is a pointer type, a number is read as a WORD and assigned to P, in a way that depends on your implementation, so that writing a pointer and later reading it yields the same pointer value. The address types should be read as WORDs using .R or .S notation.

5. String types

At the extended level, if P is a STRING (n), then the next "n" characters are read sequentially into P. Preceding line markers, spaces, tabs, or form feeds are not skipped. If the line marker is encountered before n characters have been read, the remaining characters in P are set to blanks and the file position remains at the line marker.

If the STRING is filled with n characters before the line marker is encountered, the file position remains at the next character. In a few implementations there may be a limit of 255 characters on the length of a STRING read. P can be the super array type STRING (e.g., a reference parameter or pointer referent variable).

At the extended level, if P is an LSTRING (n), then the next "n" characters are read sequentially into P, and the length of the LSTRING is set to "n." Preceding line markers, spaces, tabs, or form feeds are not skipped. If the line marker is encountered before "n" characters have been read, the length of the LSTRING is set to the number of characters read and the file position remains at the line marker.

If the LSTRING is filled with "n" characters before the line marker is encountered, the file position remains at the next character. P can be the super array type LSTRING (e.g., a reference parameter or pointer referent variable). READ (LSTRING) is handy when reading entire lines from a textfile, especially when the length of the line is needed. For example, the easiest way to copy a textfile is by using READLN and WRITELN with an LSTRING variable.

WRITE and WRITELN

PROCEDURE WRITE
PROCEDURE WRITELN

These procedures write data to textfiles. WRITE and WRITELN are defined in terms of the more primitive operation, PUT; that is, if P is an expression of type CHAR and F is a file of type TEXT, then WRITE (F, P) is equivalent to:

```
BEGIN
   F^ := P;  {Assign P to buffer variable F^}
     PUT (F) {Assign F^ to next component of file}
END
```

WRITE can take more than one parameter, as in WRITE (F, P1, P2,... Pn). This is equivalent to the following:

```
BEGIN
   WRITE (F, P1);
   WRITE (F, P2);
        .
        .
   WRITE (F, Pn)
     END
```

The procedure WRITELN writes a line marker to the end of a line. In all other respects, WRITELN is analogous to WRITE. Thus, WRITELN (F, P1, P2, ... Pn) is equivalent to:

```
BEGIN
   WRITE (P1, P2, ... Pn);
   WRITELN (F)
END
```

If either WRITE or WRITELN has no file parameter, the default file parameter is OUTPUT. Therefore, the first statement in each of the following pairs is equivalent to the second:

```
WRITE (P1, P2, ... Pn)
WRITE (OUTPUT, P1, P2, ... Pn)

WRITELN (P1, P2, ... Pn)
WRITELN (OUTPUT, P1, P2, ... Pn)
```

At the standard level, parameters in a WRITE can be expressions of any of the following types:

```
CHAR          BOOLEAN
INTEGER       STRING
REAL
```

At the extended level, expressions can also be of the following types:

```
WORD          an enumerated type
INTEGER4      a pointer type
LSTRING
```

The parameters may take optional M and N values. Although the procedure WRITE can also write to a BINARY file (i.e., not a textfile), this is not recommended for DIRECT files after a SEEK operation, because the complementary READ form does not work as you might expect. For BINARY files, WRITE (F, X) is equivalent to:

```
BEGIN
   F := X;
   PUT (F)
END
```

The form WRITE (F, P1, P2, ... Pn) is also acceptable. BINARY writes do not accept M and N values.

WRITE Formats

In textfiles, data parameters to WRITE and WRITELN may take one of the following forms:

```
P     P:M     P:M:N     P::N
```

The M and N values can be considered value parameters of type INTEGER and are used for formatting in various ways. The extended level permits M and N values for WRITEs, and permits giving N without M, as in:

```
P::N
```

Using them in a nonstandard way is an error not caught at the standard level. In some cases only M, or N, or neither, is actually used; unused M and N values are ignored.

Omitting M or N is the same as using the value MAXINT. For example, WRITE (12:MAXINT) uses the default M value (8 in this case). M and N values are not accepted for BINARY files. In WRITE, the M value is the field width used as the number of characters to write. In ISO-Pascal, M must be greater than zero,

and if the expression being written requires less than M characters, then it is padded on the left with spaces.

At the extended level, M can also be negative or zero. If it is negative, the absolute value of M is used, but padding of spaces occurs on the right instead of the left. If it is zero, no characters are written. These are ISO standard errors not caught in this Pascal.

If the representation of the expression cannot fit in ABS (M) character positions, then extra positions are used as needed for numeric types, or the value is truncated on the right for string types. If M is omitted or equal to MAXINT, a default value is used.

The N value signifies:

1. the number of decimal places if P is of type REAL

2. the output radix if P is of type INTEGER, WORD, INTEGER4, or pointer

3. the numeric or identifier value if P is of an enumerated type

Most of the following formatting rules apply to the function ENCODE as well.

1. INTEGER and WORD types

   If P is of type INTEGER, WORD, or a subrange thereof, then the decimal representation of P is written on the file. If P is a negative INTEGER, a leading minus sign is always written. WORD values are never negative. For INTEGER and WORD values, the default M value is 8.

   If ABS (M) is smaller than the representation of the number, additional character positions are used as needed. N is used to write in hexadecimal, decimal, octal, binary, or other base numbering using N equal to a number from 2 to 36; this is an extension to the ISO standard. If N is not 10 (or omitted or MAXINT), then padding on the left is with zeros and not spaces. Omitting N or setting N to MAXINT or 10 implies a decimal radix.

   WORD decimal numbers from 32768 to 65535 are written normally and not in their negative integer equivalents. All values written should be separated by spaces or some other character not valid in numbers, so that values are read as separate numbers.

2. REAL and INTEGER4 types

If P is of type REAL, a decimal representation of the number P, rounded to the specified number of decimal places, is written on the file. If the N is missing or equal to MAXINT, a floating-point representation of P is written to the file, consisting of a coefficient and a scale factor. If N is included, a rounded fixed point representation of P is written to the file, with N digits after the decimal point. If N is zero, P is written as a rounded integer, with a decimal point. The default value of M for REAL values is 14.

The following are examples of WRITE operations on REAL values:

Statement	**Output**
WRITE (123.456)	' 1.2345600E+02'
WRITE (123.456:20)	' 1.23456000000000E+02'
WRITE (123.456::3)	' 123.456'
WRITE (123.456:2:3)	' 123.456'
WRITE (123.456:-20:3)	'123.456 '

At the extended level, if P is of type INTEGER4, the decimal representation of P is written on the file. The N value is used to set the radix, as in type INTEGER. The default M value is 14.

3. Enumerated and Boolean types

   At the extended level, if P is an enumerated type and N is omitted or equal to MAXINT then ORD (P) is written on the file, as if it were a WORD. If N is given with the value 1, the enumerated type's constant identifier for the value of P is written on the file, as if it were a STRING. Note that using this N notation causes memory to be allocated for the enumerated type's constant identifiers.

   At the standard level, if P is of type BOOLEAN, then one of the strings 'TRUE' or 'FALSE' is written to the file as a STRING. The ORD value is never written for BOOLEAN types as it is for enumerated types (although you can use WRITE(ORD(P)) instead).

4. Reference types

   At the extended level, if P is a pointer type, then P is written as a WORD. This is done in an implementation defined way such that writing a pointer and later reading it produces the same pointer value. The address types should be written as WORD values using .R or .S notation.

5. String types

   If P is of type STRING (n), then the value of P is written on the file. The default value of M is the length of the STRING, "n." If ABS (M) is less than the length of the string, then only the first ABS (M) characters are written. If M is zero, nothing is written. The right portion of the STRING is always truncated, even if M is negative. In a few implementations, there may be a limit of 255 characters on the length of a STRING write.

   At the extended level, if P is of type LSTRING (n), then the value of P is written on the file. The default value of M is the current length of the string, P.LEN. If ABS (M) is less than the current length, then only the first ABS (M) characters are written. If M is zero, then nothing is written. The right portion of the LSTRING is always truncated, even if M is negative. If ABS (M) is greater than the current length, spaces, not characters, fill the remaining positions past the length in the LSTRING. Note that a string of M blanks can be written with NULL:M.

EXTENDED LEVEL I/O

The following additional I/O features are available at the extended level:

1. You can access three FCB fields: F.MODE, F.TRAP, and F.ERRS.

2. A number of additional procedures are predeclared.

3. Temporary files are available.

"Extended Level I/O," in chapter 6, Data Types discusses FCB fields in the context of files. The additional procedures and temporary files are described below.

EXTENDED LEVEL PROCEDURES

Procedure ASSIGN (VAR F; CONST N: STRING);

This procedure assigns an operating system filename in a STRING (or LSTRING) to a file F. As a rule, ASSIGN truncates any trailing blanks. ASSIGN overrides any filename set previously. A filename must be set before the first RESET or REWRITE on a file. ASSIGN on an open

file (after RESET or REWRITE but before CLOSE) produces an error. ASSIGN to INPUT or OUTPUT is allowed, but since these two files are opened automatically, they must be closed before being assigned to.

Procedure CLOSE (VAR F);

This procedure performs an operating system close on a file, ensuring that the file access is terminated correctly. This is especially important for file variables allocated on the stack or the heap. Since these files must be closed before a RETURN or DISPOSE loses the file control block, they are closed automatically when a RETURN or DISPOSE releases stack or heap file variables.

File variables with the STATIC attribute in procedures and functions are also closed automatically when the procedure or function returns. Files allocated statically at the program, module, or implementation level are automatically closed when the entire program terminates.

If necessary, when a CLOSE is executed, a file being written to has its operating system buffers flushed. However the buffer variable is not PUT. If a file of type TEXT is being written and the last nonempty line does not end with a line marker, one is added to the end of the last line. If the file has the mode SEQUENTIAL and is being written, an end-of-file is written.

Note that some runtime errors may remove control from the Pascal runtime system. In these cases, files being written may not be closed, and the information in them may be lost. A CLOSE on a file that is already closed or never opened (no RESET or REWRITE) is permitted. CLOSE is not ignored if error trapping is on and there was a previous error. CLOSE turns off error trapping for the file, and clears the error status if no errors were found.

Procedure DISCARD (VAR F);

This procedure closes and deletes an open file. DISCARD is much like CLOSE except that the file is deleted.

Procedure READFN (VAR F: P1, P2...PN);

This procedure is the same as READ (not READLN) with two exceptions:

1. File parameter F should be present (INPUT is assumed, but a warning is given if F is omitted).

2. If a parameter P is of type FILE, a sequence of characters forming a valid filename is read from F and assigned to P in the same manner as ASSIGN.

Parameters of other types are read in the same way as the READ procedure.

Note that READFN is like READ, not like READLN, and does not read the trailing line marker. If the first parameter in a READFN call is a file of any type, it is assumed to be the textfile from which characters are read. It is not assumed that the file's name should be read using INPUT as the default source.

READFN is used internally to read a program's parameters. It is useful when reading a filename and assigning the filename to some file in one operation.

Procedure READSET (VAR F; VAR L: LSTRING, CONST S: SETOFCHAR);

This procedure reads characters and puts them into L, as long as the characters are in the set S and there is room in L. If no file parameter is given, INPUT is assumed, as in READ and WRITE. Leading spaces, tabs, form feeds, and line markers are always skipped. Reading ceases at the first line marker, which is never in the type CHAR.

READSET, along with ENCODE, is used by the runtime system to do the formatted READ procedures, as well as to read filenames with READFN. It is handy when reading and parsing input lines for simple command scanners. The L and S parameters must reside in the default data segment.

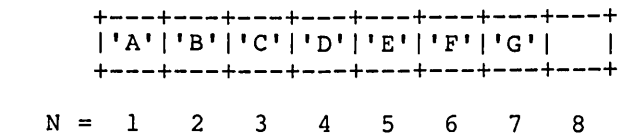
Procedure SEEK (VAR F; N: INTEGER4);

In contrast to normal sequential files, DIRECT files are random access structures. SEEK is used to randomly access components of such files. To use a DIRECT file, the MODE field must be set to DIRECT before the file is opened with RESET or REWRITE; the file, F, must be a DIRECT mode file. If the file is actually read or written sequentially, the usual READ and WRITE procedures can be used.

SEEK modifies a field in file F so that the next GET or PUT applies to record number N. The record number parameter N can be of type INTEGER or WORD, as well as

of type INTEGER4. For textfiles (ASCII structure), records are lines; for other files (BINARY structure), records are components. Record numbers start at one (not zero). If F is an ASCII file, SEEK sets the lazy evaluation status "empty." If F is a BINARY file, SEEK waits for I/O to finish and sets the concurrent I/O status "ready."

SEEK is best illustrated by some examples. Assume for instance, that a BINARY structured, DIRECT mode file contains the following CHAR contents:



An implicit SEEK 1 is done after a REWRITE or a RESET. Thus, with DIRECT mode files, the following sequences of commands might be given:

```
RESET (F);     {Initial SEEK 1, followed by GET; F^
                    now holds 'A'}

SEEK (F, 5); {File position set to 5; F^ still
                    holds 'A'}

C := F^        {C is now equal to 'A'; C does not
                    equal 'E'}
```

Note that the fifth component is not assigned to C, as you might expect. To obtain this value, the following sequences of commands should be executed:

```
RESET (F);     {Initial SEEK 1, followed by GET; F^
                    now holds 'A'.}
SEEK (F, 5); {File positioned at 5.}

GET (F);       {File buffer variable is loaded with
                    'E'.}

C := F^        {C gets value 'E'.}
```

The rule to follow is to always follow a SEEK (F, N) with a GET to assure that the nth component is contained in the buffer variable.

GET and PUT operate normally on DIRECT mode files with

BINARY structured files. However, READ and WRITE work only with ASCII files, i.e., textfiles. READ, in particular, will not work with DIRECT mode BINARY files, because it assigns the buffer variable's value before it performs a GET. On the other hand, GET and PUT are not normally used with ASCII structured DIRECT mode files. Lazy evaluation makes READ and WRITE more appropriate. Care should always be taken when mixing normal sequential operations with DIRECT mode SEEK operations.

TEMPORARY FILES

Sometimes a program needs a "scratch" file for temporary, intermediate data. If this is the case, you may create a temporary file that is independent of the operating system. To do so, without having to give the file a name in a specific format, ASSIGN a zero character as the name of the file. For example:

```
ASSIGN (F, CHR (Ø))
```

The file system creates a unique name for the file when it sees that the zero character has been assigned as a name. In environments where several running jobs are sharing a file directory, the job number is usually part of the name. Temporary files are deleted when they are closed, either explicitly or when the file gets deallocated. RESET and REWRITE do not delete the file.

CHAPTER 13
COMPILANDS

CONTENTS

A Compiland is a source file capable of being compiled by the compiler. Pascal permits three kinds of compilands: programs, modules, and implementations of units. Use of modules and implementations of units allows you to create separately compiled routines that can be linked to a program without re-compilation.

Example of a compilable program:

```
PROGRAM MAIN (INPUT, OUTPUT);
BEGIN
  WRITELN('Main Program')
END. {Main}
```

Example of a compilable module:

```
MODULE MOD_DEMO;  {No parameter list in heading}

  PROCEDURE MOD_PROC;
  BEGIN
    WRITELN ('Output from MOD_PROC in MOD_DEMO.')
  END;
END. {Mod_Demo}
```

Example of a compilable unit:

```
INTERFACE;
UNIT UNIT_DEMO (UNIT_PROC); {UNIT_PROC is the only
                             exported identifier}
  PROCEDURE UNIT_PROC;
END;
IMPLEMENTATION OF UNIT_DEMO;
  PROCEDURE UNIT_PROC;
  BEGIN
    WRITELN ('Output from UNIT_PROC in UNIT_DEMO.')
  END;
END. {Unit_Demo}
```

If you compile MODULE MOD_DEMO and UNIT UNIT_DEMO separately, you can later incorporate them into the main program as shown below:

```
INTERFACE;          {INTERFACE required at the start of any}
                    {source that implements or uses a unit.}
  UNIT UNIT_DEMO (UNIT_PROC);
  PROCEDURE UNIT_PROC;
END;

PROGRAM MAIN (INPUT, OUTPUT);

USES UNIT_DEMO;  {USES clause below needed to connect}
                 {implementation and program.}
```

```
PROCEDURE MOD_PROC; EXTERN;  {EXTERN declaration needed to
                              connect module's procedure.}
BEGIN
  WRITELN('Output from Main Program.');
  MOD_PROC;
  UNIT_PROC;
END.                         {End of main program.}
```

When the program MAIN is compiled, the output consists of the following:

1. output from Main Program

2. output from MOD_PROC declared in MOD_DEMO

3. output from UNIT_PROC declared in UNIT_DEMO

The rules governing the construction and use of programs, modules, and units are discussed in the following sections:

PROGRAMS

Except for its heading and the addition of a period at the end, a Pascal program has the same format as a procedure declaration. The statements between the keywords BEGIN and END are called the body of the program.

Example of a program:

```
{Program heading}
PROGRAM ALPHA (INPUT, OUTPUT, A_FILE, PARAMETER);

{Declaration section}
VAR A_FILE: TEXT;  PARAMETER: STRING (1Ø);

{Program body}
BEGIN
  REWRITE (A_FILE);
  WRITELN (A_FILE, PARAMETER);
END.
{Ends with period (.)}
```

The word "ALPHA" following the reserved word "PROGRAM" is the program identifier. The program identifier becomes the identifier for a parameterless PUBLIC procedure, at a scope above all other identifiers in the program. This procedure also has the PUBLIC identifier ENTGQQ, which is called during initialization to start program execution.

You could call the program body as a PUBLIC procedure from

another program, or from a module or unit, using the program identifier or ENTGQQ as the procedure name (but doing so is not recommended). This means that you can redeclare the program identifier within a program, and the usual scoping rules apply. The program identifier is at the same level as the predeclared identifiers, so giving a program an identifier like INTEGER or READ generates an error message.

The program parameters denote variables that are set from outside the program. The program communicates with its environment through these variables.

At the standard level, all variables of any FILE type should be present as program parameters, since there is no other way to give an operating system filename to the file. However, at the extended level, you may use the ASSIGN and READFN procedures to assign filenames, so file variables need not appear as program parameters.

Program parameters differ entirely from procedure parameters; they are not passed as parameters to the procedure that is the body of the program. All program parameters must be declared in the variable declaration part of the block constituting the program. If there are no program parameters and the files INPUT and OUTPUT are not referenced, you could use the following form instead:

```
PROGRAM identifier;
```

The two standard files INPUT and OUTPUT receive special treatment as program parameters. Their values are not set like other program parameters and should not be declared, since they are already predeclared. Each should be present as a program parameter if used either explicitly or implicitly in the program:

```
WRITE (OUTPUT, 'Prompt: ');    {Explicit use}
READLN (INPUT, P);

WRITE ('Prompt: ')             {Implicit use}
READLN (P);
```

The compiler gives a warning if you use them in the program but omit them as program parameters. The only effect of INPUT and OUTPUT as program parameters is to suppress this warning.

You may redefine the identifiers INPUT and OUTPUT. However, all textfile input and output procedures and functions (READ, EOLN, etc.) still use the original definition. RESET (INPUT) and REWRITE (OUTPUT) are generated automatically, whether or not they are present as program parameters; you may also generate them explicitly.

Program initialization gives a value to every program parameter variable, except INPUT and OUTPUT. Each parameter must be either

of a simple type or of a STRING, LSTRING, or FILE type (i.e., any type accepted by the READFN procedure). Program parameters must be entire variables: no component selection is permitted.

Internally, each program parameter uses the file INPUT and generates READFN calls. Before each parameter is read, a special call is made to the internal routine PPMFQQ. PPMFQQ gets characters returned from an operating system interface routine called PPMUQQ, which gets them from the command line. PPMFQQ then effectively puts those characters at the start of the file INPUT. The identifier of the parameter is passed to both routines (PPMFQQ and PPMUQQ).

MODULES

Modules provide a simple, straightforward method for combining several compilable segments into one program. Units provide a more powerful and structured method for achieving the same end.

Basically, a module is a program without a body. The identifier in the module heading has the same scope as a program identifier. The heading can also include attributes that apply to all procedures and functions in the module. There are no module parameters; nor is there a module body. A module ends with the reserved word END and a period.

Example of a module:

```
MODULE BETA [PUBLIC];          {Optional attributes}

PROCEDURE GAMMA;
   BEGIN WRITELN ('Gamma') END;

FUNCTION DELTA: WORD;
   BEGIN DELTA := 123 END;

END.                           {No body before END}
```

After the module identifier, you may give one or more attributes (in brackets) to apply to all of the procedures and functions nested directly in the module. Depending on which, if any, attributes you specify, the following assumptions or restrictions apply:

1. If there is no attribute list at all, the PUBLIC attribute is assumed. However, if a list is present but empty, PUBLIC is not assumed.

2. The EXTERN directive used with a particular procedure or function overrides the PUBLIC attribute given (or assumed) for the entire module.

3. EXTERN and ORIGIN cannot be given as attributes for an entire module, although you may specify them for individual procedures and functions.

4. If PURE is used, the module must contain only functions for PURE.

5. PUBLIC is the default attribute for all procedures and functions. However, in some cases, a PUBLIC procedure call has more overhead than a purely local one. In other cases, the identifier of a local procedure may conflict with a global identifier passed to the linker. To avoid these problems, use PUBLIC with selected individual procedures and functions and empty brackets for the entire module (e.g., MODULE BETA [];).

Although a module contains no body, only declarations, you may use it as a parameterless procedure; that is, you may declare the module identifier as a procedure and call it from other programs, modules, or units. This module procedure (unlike a similar procedure for programs or units) is never called automatically, since there is no way for the compiler to know whether a module has been loaded and thus whether to generate a call to it.

However, in some cases, the compiler generates module initialization code that should be executed by calling the module as an EXTERN procedure. If such code is necessary, the compiler gives the warning:

```
Initialize Module
```

If you see this message, declare the module as a parameterless EXTERN procedure and call the procedure once before anything in the module is accessed. (You will need to do this if module declares any FILE variables.)

Given a module M that declares its own file variables, a program that uses M should look like this:

```
PROGRAM P (INPUT, OUTPUT)
  .
  .
PROCEDURE M; EXTERN;
BEGIN
  M;              {Runtime call initializes}
  .               {file variables.}
  .
END.
```

If the module USES any interfaces that require initialization, the compiler generates a warning that you should declare the module EXTERN and call it as described in the previous paragraph.

If module M does not contain any of its own file variables or use any initialized units, there is no need to invoke M as a procedure in the body of the program or to declare it as an EXTERN procedure.

Variables within modules are not automatically given any attributes. Except for the initialization of FILE variables mentioned above, variables within modules are the same as program variables.

UNITS

Units provide a structured way to access separately compiled modules. A unit has two parts:

1. an interface
2. an implementation

The interface appears at the front of an implementation of a unit and at the front of any program, module, interface, or implementation that uses a unit.

A unit contains constants, types, super types, variables, procedures, and functions, all of which are declared in the interface of the unit. Any program, module, or implementation or another interface may use an interface. An implementation contains the bodies of the procedures and functions in a unit, as well as optional initialization for the unit.

When you are using units, their interfaces go before everything else in a source file, either in an IMPLEMENTATION or in the program, module, or other unit that uses it. By separating the interface from the implementation, you can write and compile a program before or while writing the implementation. Or, you may load a program with one of several implementations (for example, one in Pascal or one in assembly language).

A large Pascal program is often better organized as a main program and a number of units. However, only a program, module, interface, or implementation can USE a unit, not an individual procedure or function.

A program, module, implementation, or interface that uses an interface must start with the source file for that interface. Generally, the interface source file is a separate file, and an $INCLUDE metacommand at the start of the source file brings in the interface source itself at compile time. Because there is then only one master copy of the interface, this is easier and

more reliable than physically inserting the interface everywhere it is used (and running the risk of ending up with several different versions).

In some applications, you may wish to have several versions of the same interface. For example, there is a separate version of the file control block interface for every target file system; the $INCLUDEd file is copied from the desired interface version before the program using it is compiled. Naturally, every version must declare the common identifiers; each might also have some constant values for use in $IF metacommands for the version-specific portions of the interface.

A source file of any kind contains zero or more unit interfaces, separated by semicolons, and followed by a program, a module, or an implementation, which is followed by a period. Each of these entities is called a "division." See "The Interface Division," and "The Implementation Division," in this chapter for details about divisions.

A unit consists of the unit identifier, followed by a list of identifiers in parentheses. These identifiers are called the constituents of the unit and are the ones provided by a unit or required by a program, module, or other unit. The unit is preceded by the keyword UNIT for a provided unit or USES for a required one.

All unit identifiers in a source file must be unique. The identifiers in parentheses, however, may differ in the providing and requiring divisions. Correspondence between identifiers provided and required is by position in the list (similar to formal and actual parameters in procedures).

The identifier list in a USES clause is optional; if not given, the identifiers in the UNIT list are used by default. Giving different identifiers in a USES clause allows you to change the identifiers in case several different interfaces have identifier conflicts. Multiple USES clauses can be combined; thus, the following statements are equivalent:

```
USES A; USES B; USES C;
USES A, B, C;
```

Note also that a unit may introduce optional initialization code. Such code is implied by the words BEGIN and END at the end of an interface and is provided in an optional body in an IMPLEMENTATION.

Example of a unit that introduces initialization code:

The program file, PLOTBOX:

```
{$INCLUDE:'GRAPHI'}
PROGRAM PLOTBOX (INPUT, OUTPUT);
```

```
  USES GRAPHICS (MOVE, PLOT);
  {MOVE and PLOT are USEd identifiers.}
  BEGIN
    MOVE (Ø, Ø);
    PLOT (1Ø, Ø);  PLOT (1Ø, 1Ø);
    PLOT (Ø, 1Ø);  PLOT (Ø, Ø);
  END.
```

The interface file, GRAPHI:

```
INTERFACE;
  UNIT GRAPHICS (BJUMP, WJUMP);
  {Exported identifiers are BJUMP and WJUMP.}
  {In the above PROGRAM, MOVE and PLOT}
  {are aliases for these identifiers.}
  PROCEDURE BJUMP (X, Y: INTEGER);
  PROCEDURE WJUMP (X, Y: INTEGER);
  {Procedure headings only above.}
 BEGIN
 {BEGIN implies initialization code.}
 END;
```

The implementation file:

```
{$INCLUDE:'GRAPHI'}
{$INCLUDE:'BASEPL'}
{The following implementation USES}
{the UNIT BASEPL.  Thus, the interface}
{is included above and the unit}
{used below.}
IMPLEMENTATION OF GRAPHICS;
{Implementation is invisible to user.}
  USES BASEPLOT;
    {Procedures BJUMP and WJUMP are}
    {implemented below.}
    {Note that only the identifiers}
    {are given in the heading.}
    {The parameter lists are given}
    {in the interface.}
  PROCEDURE BJUMP;
    BEGIN DRAWLINE (BLACK, X, Y) END;
  PROCEDURE WJUMP;
    BEGIN DRAWLINE (WHITE, X, Y) END;
BEGIN
{Begin initialization.}
  DRAWLINE (BLACK, Ø, Ø)
END.
```

The interface file, BASEPL:

```
INTERFACE;
  UNIT BASEPLOT (BLACK, WHITE, DRAWLINE);
   {Other identifiers besides procedure}
   {identifiers can be exported.}
```

```
    {Note that BLACK and WHITE are}
    {exported constant identifiers.}
  TYPE RAINBOW = (BLACK, WHITE, RED, BLUE, GREEN);
  PROCEDURE DRAWLINE (C: RAINBOW;  H, V: INTEGER);
{No BEGIN; therefore, not an initialized unit.}
END;
```

A USES clause may occur only directly after a program, module, interface, or implementation heading. When the compiler encounters a USES clause, it enters each constituent identifier (from the UNIT clause or USES clause itself) in the symbol table. Identifiers for variables, procedures, and functions are associated with the corresponding identifiers in the interface, which then become external references for the linker.

If the sample program above were compiled, every reference to the procedure PLOT would generate an external reference to WJUMP. However, references to DRAWLINE would use the same identifier for the external reference.

Constants and types (including any super array types) in the interface are simply entered in the program's symbol table (along with the new identifier, if any). Thus, a type in an interface is identical to the corresponding type in the USES clause.

Record field identifiers are the same in the program, interface, and implementation. Enumerated type constant identifiers must be given explicitly, if needed; they are not automatically implied by the enumerated type identifier. Labels cannot be provided by an interface, since the target label of a GOTO must occur in the same division as the GOTO.

The Interface Division

The structure of an interface is as follows:

1. An interface section starts with the reserved word INTERFACE, an optional version number in parentheses, and a semicolon.

2. Next comes the keyword UNIT, the unit identifier, the parenthesized list of exported constituent) identifiers, and another semicolon.

3. Any other units required by this interface come next, in USES clauses.

4. The last section is the actual declarations for all identifiers given in the interface list, using the usual CONST, TYPE, and VAR sections and procedure and function headings, in any order. No LABEL or VALUE sections are permitted.

5. The interface ends with BEGIN END if it has initialization, or just with END if it has no initialization.

Except for ORIGIN, which cannot currently be used in interfaces, most available attributes can be given to variables, procedures, and functions. Because the PUBLIC or EXTERN attribute or EXTERN directive is given automatically, you must not specify attributes that may conflict (e.g., PUBLIC and EXTERN).

Usually the only identifiers you can declare are the constituents, but other identifiers are permitted. If the interface needs a call to initialize the unit, the keyword BEGIN generates the call. The interface ends with the reserved word END and a semicolon.

Example of an interface division:

```
INTERFACE (3);
  UNIT KEYFILE (FINDKEY, INSKEY, DELKEY, KEYREC);
    USES KEYPRIM (KFCB, KEYREC);

    PROCEDURE FINDKEY (CONST NAME: LSTRING;
        VAR KEY: KEYREC;
        VAR REC: LSTRING);
    PROCEDURE INSKEY  (CONST REC: LSTRING;
        VAR KEY: KEYREC);
    PROCEDURE DELKEY  (CONST KEY: KEYREC);
    PROCEDURE NEWKEY  (CONST KEY: KEYREC);
  BEGIN
  {Signifies initialized unit.}
  END;
```

In this example, KEYREC is part of the unit KEYPRIM, but is exported as part of the unit KEYFILE. KFCB is also part of the KEYPRIM unit, but is not exported by the KEYFILE unit. NEWKEY is defined in the interface, but not exported by the KEYFILE unit. This is permitted, but is pointless, since NEWKEY is unknown even in the the implementation of the unit.

Memory available at compile time limits the number of identifiers the compiler can process. This limit can be a problem if you have many interfaces, especially interfaces that use other interfaces. The symptom is the following error message:

```
Compiler Out Of Memory
```

The message occurs before the final USES clause in the program, module, or implementation you are compiling. The cure is to reduce the number of identifiers in interfaces USEd by other interfaces. For example, make a single interface that contains only types (and type-related constants) shared by your other interfaces, and only USE this interface in the others.

If you include any file variables in the interface, the unit must be initialized. The compiler does not give the usual warning,

```
Initialize Variable
```

when you declare a file in an interface. If your interface contains files, be sure to end it with BEGIN END so that it will be initialized.

The Implementation Division

You can compile an implementation of a unit separately from other programs, modules, or units, but you must compile it along with its interface.

The structure of an implementation is as follows:

1. An implementation of an interface starts with the reserved words IMPLEMENTATION OF, followed by the unit identifier and a semicolon.

2. Next comes a USES clause for units it needs only for its own use.

3. Then comes the usual LABEL, CONSTANT, TYPE, VAR, and VALUE sections and all procedures and functions mentioned as constituents (which must be in the outer block) or used internally, in any order.

VALUE and LABEL sections may appear in the implementation, but not in the interface.

Example of an implementation:

```
IMPLEMENTATION OF KEYFILE;
  USES KEYPRIM (KEYBLOCK, KEYREC);

  VAR KEYTEMP: KEYREC;

  PROCEDURE FINDKEY;
    BEGIN
      .
      {Code for FINDKEY}
      .
    END;

  PROCEDURE INKEY;
    BEGIN
      .
      {Code for INKEY}
```

```
    .
  END;

PROCEDURE DELKEY;
  BEGIN
    .
    {Code for DELKEY}
    .
  END;

BEGIN
  .
  {Any initialization code goes here.}
  .
END.
```

Constants, variables, and types declared in the interface are not redeclared in the implementation. However, you may declare other "private" ones. Procedures and functions that are constituents of the unit do not include their parameter list (it is implied by the interface) or any attributes. (The PUBLIC attribute is implied, unless the EXTERN directive is given explicitly.)

All procedures and functions in the INTERFACE must be defined in the IMPLEMENTATION. However, they can be given the EXTERN directive so that several IMPLEMENTATIONs (or an IMPLEMENTATION and assembly code) can implement a single INTERFACE. All procedures and functions with the EXTERN directive must appear first; the compiler checks for this and issues an error message if the EXTERN directive is missing or misplaced.

You may implement a unit in assembly language, in which case all variables, procedures, and functions should generate public definitions for the loader. If the interface is not implemented in Pascal, it must give the proper calling sequence attribute (and of course you must be familiar with calling sequences and internal representation of parameters).

Several Pascal runtime units are implemented partially in Pascal and partially in assembly language. As mentioned, any IMPLEMENTATION section that does not implement all interface procedures and functions must, at the start of the IMPLEMENTATION, declare such procedures and functions to be EXTERN.

An implementation, like a program, may have a body. The body is executed when the program that uses the unit is invoked, so any initialization needed by the unit can be done. This includes internal initialization, such as file variable initialization, as well as user initialization code. If the source file contains several units, each implementation body is called in the order its USES clause appears found in the source file. However, initialization code for a unit is executed only once, no matter

how many clauses refer to it.

The body, as in a program, is a list of statements enclosed with the reserved words BEGIN and END. At initialization time, the version number of the interface with which the implementation was compiled is compared against the version number of the interface with which the program was compiled. These must be the same. This checking prevents you from trying to run a program with obsolete implementations. If no version number is given, zero is assumed.

The keyword BEGIN before the final END indicates a unit with initialization. If the word BEGIN is omitted, the implementation must not have a body and no initialization takes place. Uninitialized units lack the following:

1. user initialization code

2. a guarantee of only one initialization

3. a version number check

The format for an initialized implementation of a unit is similar to a program:

```
IMPLEMENTATION OF unit-identifier
declarations
BEGIN
   body                {Initialization code}
END.
```

The format for an uninitialized implementation of a unit is similar to a module:

```
IMPLEMENTATION OF unit-identifier
 declarations
                       {No initialization code}
END.
```

If the implementation for an uninitialized unit declares any files or USES any interfaces that require initialization, the compiler warns you to initialize the implementation. Initialization is done automatically if you add the keyword BEGIN to both the interface and the implementation. As with a module, you can declare an uninitialized unit to be a procedure with the EXTERN attribute and then initialize it by calling it from the program.

CHAPTER 14
COMPILING, LINKING, AND EXECUTING PROGRAMS

CONTENTS

INVOKING THE PASCAL COMPILER FROM THE EXECUTIVE

Invoke the Pascal compiler with the Executive's Pascal command. The following form appears:

```
Pascal
     Source File          _____________________
     [Object file]        _____________________
     [List file]          _____________________
     [Object list file]   _____________________
```

Filling in a form is described in the B 20 System Software Operation Guide.

FIELD DESCRIPTIONS

Source file

is the name of the Pascal source file to be compiled.

[Object file]

is the name of the destination file for the object code that results from the compilation. If no file is specified, a default object file is chosen as follows. Pascal treats the source name as a character string, strips off any suffix beginning with the character period (.), and adds the characters ".Obj". The result is the name of the file. For example, if the source file is:

```
[Dev]<Jones>Main
```

then the default object file is:

```
[Dev]<Jones>Main.Obj
```

If the source file is:

```
Prog.Pas
```

then the default object is:

```
Prog.Obj
```

[List file]

is the name of the file to which to write a listing of the compilation. If no file is specified, the default list file is chosen in a manner similar to the default object file, except that the string added is ".Lst" instead of ".Obj". To list portions of the list file, see the $LIST metacommand.

[Object list file]

is the name of the file to which to write the listing of the generated object code. If no file is specified, the default is to suppress the generation of the object list file. To list portions of the object list file, see the $OCODE metacommand.

LINKING A PASCAL PROGRAM

Invoke the Linker as described in the Linker/Librarian Manual. The following special features of the Linker as applied to Pascal are important:

[Libraries]

When linking a Pascal program, the Linker automatically searches the library (SYS)<Sys>Pascal.Lib (if it exists) for any unresolved external symbols.

[DS allocation?]

When linking a Pascal program, the Linker automatically assumes "Yes" for this field. Most Pascal applications require (DS allocations?) to be "Yes." To link an application involving a Pascal module without (DS allocation?), "No" must be explicitly specified for this field.

RUNNING A PASCAL PROGRAM

Once a run file has been obtained through use of the Linker, a Pascal program can be run either by using the Executive's Run command, or by creating a customized command using the Executive's New Command command and invoking it. The latter approach allows fields in the command form for the customized command to be passed to Pascal program parameters declared in the Pascal program header. For example, when the following program is used in conjunction with an Executive command having two fields, the video display shows the contents of these fields:

```
Program ReadParam(OUTPUT, field1, field2);

VAR       field1, field2  :  LSTRING (255);

BEGIN
          WriteLn(field1);
          WriteLn(field2);
END.
```

RUNTIME SIZE AND DEBUGGING

The Pascal runtime occupies approximately 67K bytes: 10K of data (including the stack) and 57K of code (which implements the Pascal files system, heap, error handling, encode/decode, and includes SAM and DAM from Btos.lib). (included in the 57K is memory for Reals, Sets, and LStrings, which occupy 7.5K, 2.5K, and 2K, respectively.) If you do not use these facilities, you may suppress their inclusion in your program by explicitly linking in the module Pasmin.obj--you will then have to do all memory management, input-output, etc. by calls to BTOS facilities. If you do link in PasMin.obj, you may set [DS Allocation?] to either Yes or No.

All Pascal static data, including the 10K of system static data mentioned above, is limited to 64K. Therefore you can have at most 54K of user static data and heap. You can dynamically allocate data above this limit by calling the BTOS memory allocation primitives directly and addressing the memory so-allocated with ADS pointers. Since the Pascal heap is allocated dynamically in short-lived storage and must be contiguous, once you allocate short-lived memory the heap cannot grow larger. The function PreAllocHeap allows the user the ability to pre-allocate the heap. Although you can allocate and address storage with no limit other than total physical memory, no single Pascal object can exceed 64K bytes (which is the space required for an array of 16K real numbers).

Suppose you start a task that has a Pascal main program using CODE-GO so as to get control in the Debugger. Find your main program by attaching the ".sym" file created when you linked (the main program starts at ENTGQQ). If symbols are not available, you may also examine the instruction at CS:77H (if linking with the standard Pascal runtime) or at CS:1BH (if linking with the minimal Pascal runtime-- Pasmin.obj), which is a call to the main program.

COMPILING AND LINKING LARGE PROGRAMS

Occasionally, you may find that a large program exceeds one or more physical limits on the size of program the compiler, the linker, or your machine can handle. This section describes some ways to avoid or work within such limits.

Avoiding Limits on Code Size

The upper limit on the size of code that can be generated at once by the Pascal Compiler is 64K bytes. However, since you can compile any number of compilands separately and link them together later, the real program size limit is not 64K but the amount of memory available.

For example, you can separately compile six different compilands of 50K bytes each. Linking them together produces a program with a total of 300K bytes of code.

In practice, a source file large enough to generate 64K bytes of code would be thousands of lines long, and unwieldy both to edit and to maintain. A better practice is to break a large program into modules and units to better structure the development and maintenance process. As always, there is a tradeoff between size and speed. Procedure and function calls within a module to routines without the PUBLIC attribute are somewhat faster, since intrasegment calls, which run faster, are generated rather than intersegment calls.

Avoiding Limits on Data Size

Data includes your main variables, the stack, and the heap. Pascal operates with data in two regions of memory:

1. the default data segment
2. the segmented data space

The upper limit on the amount of data that can reside in the default data segment is also 64K bytes. You can go beyond this limit by taking advantage of the ability to place certain kinds of data outside the default data segment, using ADS variables, VARS and CONSTS parameters, and segmented ORIGIN variables.

The default data segment normally holds the following:

1. all statically allocated variables
2. constants that reside in memory

3. heap variables

4. the stack, which holds parameters, return addresses, stack variables, etc.

Although operations with data in the default data segment are more efficient (i.e., generate less code and run faster) than those with data that may be in any segment, almost all operations work equally well on data outside the default data segment.

The segmented data space includes the entire B 20 address space, including the default data segment. Data outside the default data segment can be referenced using ADS (segmented address) variables, VARS and CONSTS parameters, and segmented ORIGIN variables. See the appropriate chapters in this Manual for a discussion of these Pascal features.

Only in the following cases must data reside in the default data segment:

1. file variables

2. the LSTRING parameters to ENCODE and DECODE

3. all parameters to READSET

To allocate data outside the default data segment, you must go outside the Pascal system itself. If you already know the address of free blocks of memory on your computer, you can use these addresses in a segmented ORIGIN attribute or assign them to an ADS variable.

Many applications use a large block of memory for primary data, as well as various other variables to control access and processing of this data. For example, a text editor will have a work area; a data base system will have a data area (or index area); and so on. This large block can be managed outside the default data segment with ADS variables.

In the default data segment, the heap and the stack grow toward each other. Heap allocation will attempt to use existing disposed blocks in the heap itself, before growing into memory shared with the stack.

As a part of this process, adjacent disposed blocks are merged, and free blocks at the end of the heap become available to the stack.

However, only heap allocation (i.e., NEW or GETHQQ) releases free heap blocks to the stack. Therefore, if you are running out of stack after a number of DISPOSE operations, make the following call:

```
EVAL (GETHQQ (65534));
```

Working With Limits on Compile Time Memory

During compilation, large programs are most often limited in the number of identifiers in any one source file. They are occasionally limited by the complexity of the program itself. If one of these limits is reached, you will see the following error message:

```
Compiler Out Of Memory
```

There is no particular limit on number of bytes in a source file. The number of lines is limited to 32767, but in practice, any source file this big will run into other limits first.

Identifiers

Pass one of the compiler can handle a maximum of about a thousand identifiers visible at any one time. This assumes a 64K default data segment (i.e., about 160K of memory total); it also assumes that most of your identifiers are seven characters or shorter and are not PUBLIC or EXTERN.

Once a procedure or function is compiled, its local identifiers can be released to provide room for new ones. Several methods of reducing the number of identifiers in a program are described below.

1. Break your program into modules or units

   The best way to reduce the number of identifiers is to break up your program into modules or units. When dividing your application into pieces, one guiding principle is to minimize the number of shared (i.e., PUBLIC and EXTERN) identifiers. This not only is good programming practice, it makes compilation easier.

   Breaking up a program may force you to choose between a shared variable and a shared procedure or function. Usually a shared procedure or function is "cleaner"; it is easier to trace the use of a procedure than the use of a variable, for example. However, a shared variable is usually more efficient in terms of memory required and number of identifiers used.

2. Simplify your identifiers

Although it reduces the readability of a program (since naming something is a more readable way of referring to it than giving an arbitrary number), you may simplify your identifiers by replacing names with numbers. If necessary, any of the following may help:

a. Change enumerated types into WORD types and use numbers instead of identifiers.

b. Use constant literals instead of constant identifiers.

c. Combine related procedures and functions into single ones, with a parameter indicating the type of call.

d. Combine variables into an array and refer to the variables using constant array indices.

3. Remove unneeded identifiers from PASKEY

It is also possible to remove identifiers of predeclared procedures and functions you don't need from PASKEY, at least those in the final section (the one that looks like an interface). An identifier in this section must be removed three times: once from the UNIT list, once from the interface (the declaration itself), and once from the USES list. However, the type FCBFQQ must not be removed.

You can also remove identifiers of intrinsic procedures and functions, a list near the start of PASKEY from READLN to RESULT. Any identifiers removed must be replaced with an asterisk (*). However, the procedure READFN must not be removed if you have program parameters.

Finally, the following declarations can be removed:

```
ADAPQQ      INTEGER2
ADRMEM      MAXINT
ADSMEM      MAXINT4
BYTE        MAXWORD
INTEGER1    BYTE
```

Removing any other identifiers from PASKEY will generate the following error:

144 Compiler Internal Error

A special caution is required regarding interfaces. When an interface USES another interface, it must import all identifiers

in the other interface. To do this, the other interface must have been declared, so now its identifiers occur twice. If a third interface USES both of the first two, the first interface's identifiers occur three times and the second interface's identifiers occur twice, and so on. This is an easy way to run out of identifiers!

The only reason an interface needs to USE another interface is to import identifiers for types; an interface has no use for variables, procedures, and functions. You can declare a single interface with global types; this is the only interface used by other interfaces. Once compilation gets past the USES clausein the PROGRAM, MODULE, or IMPLEMENTATION, many of these "extra" identifiers are removed.

Complex Expressions

It is also possible to run out of memory in pass one with any of the following:

1. a very complex statement or expression (i.e., one that is very deeply nested)

2. a large number of error messages

3. a large number of structured constants, including string constants

You may be able to change literal strings and other structured constants into EXTERN READONLY variables which get initialized (as PUBLIC variables) in another module.

Usually, if a program gets through pass one without running out of memory, it will get through pass two. The major exception occurs with complex basic blocks, as in either of the following:

1. sequences of statements with no labels or other breaks

2. sequences of statements containing very long expressions or parameter lists (especially a WRITE or WRITELN procedure call with many expressions)

If pass two runs out of memory, it displays the following message:

```
Compiler Out Of Memory
```

The error message will give a line number reference. If there is a particularly long expression or parameter list near this line, break it up by assigning parts of the expression to local variables (or using multiple WRITE calls). If this does not work, add labels to statements to break the basic block.

LISTING FILE FORMAT

The following discussion of listing file format is keyed to this sample listing:

```
Use   Title                                          PAGE   1
User  Subtitle

JG IC Line#  Pascal 3.Ø
   ØØ    1  PROGRAM foo;    {$symtab+}
   1Ø    2  VAR i:integer; k:ARRAY [-9..Ø] OF integer;
         2  --------------------Warning 156,Assumed ;^
   2Ø    3  FUNCTION bar (VAR j: integer): integer;
   2Ø    4    VAR k: ARRAY [Ø..9] OF integer;
   2Ø    5    BEGIN
+  21    6    GOTO 1;           {jump forward}
         6  --------^Warning 281 Label Assumed Declared
 = 21    7      i := bar (j);   {assign to global}
         8      1:              {label}
 / 21    9      j := bar (i);   {global to VAR parm}
-  21   1Ø      GOTO 1;         {jump backward}
*  21   11      RETURN; GOTO 1;{other jumps}
 % 21   12      i := bar (i);   {other global reference}
   21   13      j:= bar (j);    {no global references}
   1Ø   14    END;
        14  ----^3Ø6 Function Assignment Not Found

Symtab  14  Offset Length Variable - BAR
            -     2     24 Return offset, Frame length
            -     2      2 (func'n return):Integer
            +     4      2 J               :Integer VarP
            -    22     2Ø K               :Array

   1Ø   15  BEGIN
   11   16   i := bar (i);
   ØØ   17  END.

Symtab  17  Offset Length Variable
                 Ø     24 Return offset, Frame length
                 2      2 I               :Integer
                 4     2Ø K               :Array

            Errors  Warns In Pass One
                 1      2
```

Every page has a heading that includes such information as your title and subtitle, set with the metacommands $TITLE and $SUBTITLE, respectively. If these metacommands appear on the

first source line, they take effect on the first page. The page number appears in the right side of the first line of the heading. You can set the page number with $PAGE:<n> or start a new page with $PAGE+.

The fourth line of the listing contains the column labels. The contents of the first three columns are as follows:

1. The JG column

   The JG column contains flag characters generated for your information. Jump flags, which appear under the J, may contain one of the following characters:

   + forward jump (BREAK or GOTO a label not yet encountered)

   - backward jump (CYCLE or GOTO a label already encountered)

   * other jumps (RETURN or a mixture of jumps)

   Codes for global variables (not local to the current procedure or function) appear in the column under G:

   = assignment to a nonlocal variable

   / passing a nonlocal variable as a reference parameter

   % a combination of the two

2. The IC column

   The IC column contains information about the current nesting levels. The digit under the I refers to the identifier (scope) level, which changes with procedure and function declarations, as well as with record declarations and WITH statements. The digit in the C column refers to the control statement level; this number changes with BEGIN and END pairs, as well as with CASE and END and REPEAT and UNTIL pairs. The number in this column is useful for finding missing END keywords.

   If a line is not actively used by the compiler, all these columns are blank. Thus you can locate a portion of the source accidentally commented out or skipped due

to an $IF and $END pair.

3. The Line column

The Line column shows the line number of the line in the source file. An $INCLUDEd file gets its own sequence of line numbers. If $LINE is on, this line number and the source file name identify runtime errors.

Two kinds of compiler messages appear in the listing: errors and warnings. A compilation with any errors cannot generate code. A compilation with warnings only can generate code, but the result may not execute correctly. Warnings start with the word "Warning" and a number (see, for example, line 2 in the sample listing). Errors start with an error number (see line 14 in the sample listing). See Appendix D, "Error Messages," for a complete listing of all warning and error messages.

You can suppress warning messages with the metacommand $WARN-, but this is not generally recommended. The metacommand $BRAVE+ sends error and warning messages to your terminal (as well as to the listing file). However, if there are more than will fit on a single screen, the first ones will scroll off. The location of the error is indicated in the listing file with an up arrow (^). The message itself may appear to the left or right of the arrow and is preceded with a dashed line.

Sometimes, the compiler does not detect an error until after the listing of the following line. In this case, the error message line number is not in sequence. Tabs are allowed in the source and are passed on to the listing unchanged. If the tab spacing is every eight columns, the error pointer (^) is generally correct. However, an error pointer near the end of a line may be displaced if the following line has tabs. If the compiler encounters an error from which it cannot recover, it gives the message "Compiler Cannot Continue!". This message appears if any of the following occurs:

1. The keyword PROGRAM (or IMPLEMENTATION, INTERFACE, or MODULE) is not found, or the program, module, or unit identifier is missing.

2. The compiler encounters an unexpected end-of-file.

3. The compiler finds too many errors; the maximum number of errors per page is set with the $ERRORS metacommand (the default is 25).

4. The identifier scope becomes too deeply nested.

When the compiler is unable to continue, for whatever reason, it simply writes the rest of the program to the listing file with very little error checking.

APPENDIX A
AN OVERVIEW OF THE FILE SYSTEM

This extended Pascal is designed to be easily interfaced to the operating system. The standard interface has two parts:

1. a file control block (FCB) declaration

2. a set of procedures and functions, called Unit U, that are called from Pascal at runtime to perform input and output

This interface supports three access methods: TERMINAL, SEQUENTIAL, and DIRECT.

Each file has an associated FCB (file control block). The FCB record type begins with a number of standard fields that are independent of the operating system. Following these standard fields are fields such as channel numbers, buffers, and other data, that are dependent on the operating system.

The advanced Pascal user can access FCB fields directly, as explained under Files in Chapter 6, "Data Types," of this Manual.

Pascal has two special file control blocks that correspond to the keyboard and the screen of your terminal. These two file control blocks are always available. they are the predeclared files INPUT and OUTPUT (which you can reassign and generally treat like any other files).

For files, each FCB ends with the buffer variable that contains the current file component. This means that the length of an FCB is the length of its fixed portion plus the length of the buffer variable.

File control blocks always reside in the default data segment, so they can be referenced with the offset (ADR) addresses instead of the segmented (ADS) addresses.

File variables can occur:

1. in static memory

2. on the stack as local variables

3. or in the heap as heap variables

The generated code initializes file control blocks when they are allocated and CLOSEs them when they are deallocated. For example, a fixed number of file "slots" may be available, or the routines for heap allocation may be used. A FCB can be created or destroyed, but never moved or copied.

The Compiler must know enough about a FCB to allocate one. Thus, it needs to know the length of a FCB less the length of its

buffer variable. This information is read by the compiler during initialization from a special file called PASKEY.

Unit U refers to the operating system interface routines. The file routines are called Unit F. Code generated by the compiler contains calls to Unit F which in turn calls Unit U routines.

The file system uses the following naming convention for public linker names:

1. All linker globals are six alphabetic characters, ending with QQ. (This helps to avoid conflicts with program global names.)

2. The fourth letter indicates a general class, where:

   a. xxxFQQ is part of the generic Pascal file unit.

   b. xxxUQQ is part of the operating system interface unit.

File system error conditions may be detected at the lower Unit U level, or at the higher Unit F, or undetected. When a Unit U routine detects an error, it sets an appropriate flag in the FCB and returns to the calling Unit F routine. When Unit F detects an error or discovers Unit U has detected one, it takes one of two possible actions:

1. An immediate runtime error message is generated and the program aborts.

2. Unit F returns to the calling program if error trapping has been set with the TRAP flag.

Unit F will not pass a file with an error condition to a Unit U routine. For some access methods, certain file operations may lead to an undetected error, such as reading past the end of a record (this condition has undefined results). Runtime errors that cause a program abort use the standard error-handling system, which gives the context of the error and provides entry to the debugging system.

The distributed implementation of the Pascal Compiler includes the following three source files:

1. FINU contains procedure and function headers for all Unit U routines.

2. FINK contains the common FCB declarations for all Pascal systems, along with the declaration of the FILEMODES type.

3. FINKXM contains the FCB declarations as extended for use in the BTOS environment.

APPENDIX B
COMPILER STRUCTURE

CONTENTS

The compiler is divided into three phases, or passes, each of which performs a specific part of the compilation process. Pass one, which normally corresponds to a file named PascalFe.Run, constitutes the front end of the compiler and performs the following actions:

1. reads the source program
2. compiles the source into an intermediate form
3. writes the source listing file
4. writes the symbol table file
5. writes the intermediate code file

Passes two and three (PascalOpt.Run and PascalLst.Run) together make up the back end, which does the following:

1. optimizes the intermediate code
2. generates target code from intermediate code
3. writes and reads the intermediate binary file
4. writes the object (link text) file
5. writes the object listing file

Both the front and back end of the compiler are written in Pascal, in a source format that can be transformed into either relatively standard Pascal or into system level Pascal.

All intermediate files contain Pascal records. The front and back ends include a common constant and type definition file called PASCOM, which defines the intermediate code and symbol table types. The back ends use a similar file for the intermediate binary file definition. Formatted dump programs for all intermediate files and object files are available for special purpose debugging.

The symbol table record is relatively complex, with a variant for every kind of identifier (assorted data types, variables, procedures and functions). The intermediate code (or Icode) record contains an Icode number, opcode, and up to four arguments; an argument can be the Icode number of another Icode to represent expressions in tree form, or something else (such as a symbol table reference, constant, or length). The intermediate binary code record contains several variants for absolute code or data bytes, public or external references, label references and definitions, etc.

THE FRONT END

The front end can be divided into following parts:

1. the scanner

2. low-level utilities

3. intermediate-level utilities for identifiers, symbols, Icodes, memory allocation, and type compatibility

4. high-level routines for processing procedure and function calls, expressions, statements, and declarations

The front end is driven by recursive descent syntax analysis, using a set of procedures such as EXPR (for expressions), STATEMT (for statements), and TYPEDEC (for type declarations).

The front end maintains a "current" symbol and a "look ahead" symbol. While not necessary for parsing correct programs, these symbols are useful for error recovery. Syntax errors are processed by a procedure that forces the current symbol to one of a set of symbols legal at a given point. If the current symbol is wrong, but the following one is correct, the current symbol is deleted. In all cases the correct symbol is inserted if possible. However, common substitution mistakes, such as confusing (=) and (:=), cause only a warning message to be given during compilation.

The scanner is relatively large, since it must process metalanguage and produce a listing with error messages, data about variables, and other information for the user.

Intermediate code is written to the Icode file on disk as soon as it is generated: there is no reason to keep it in memory. The symbol table is built as a binary tree of identifiers with pointers to semantic records. At the end of each block, all new semantic records are written to the symbol table file. When an error is detected, all writing to intermediate files stops, since the code may not be acceptable to the back end. Detecting a warning, rather than an error, does not invalidate the intermediate files.

The front end reads a file called PASKEY to initialize predeclared identifiers such as INTEGER, READ, and MAXINT. PASKEY can be divided into four sections.

1. The first contains the number of bytes in a file control block and the primitive type identifiers.

2. The second section lists all the intrinsic procedure and function identifiers (those that are transformed by the

front end in special ways).

3. The third section contains constants, types, and external procedures and functions using normal Pascal syntax.

4. The fourth contains one or more INTERFACE and USES clauses for predeclared procedures and functions.

THE BACK END

Of the separate passes that make up the back end of the compiler, pass two is required while pass three is optional. Pass two produces the object file, while pass three produces the object listing.

Pass Two

The optimizer reads the interpass files in the following order: first the symbol table for a block is read; then the intermediate code for the block. Optimization is performed on each "basic block, " that is, each block of intermediate code up to the first internal or user label or up to a fixed maximum number of Icodes, whichever comes first.

Within this block, the optimizer can reorder and condense expressions so long as the intent of the program(mer) is preserved. For instance, in the following program fragment, the array address A [J, K] need be calculated only once.

```
A [J, K] := A [J, K] + 1;
{J := J - 1;}
IF A [J, K] = MAX THEN PUNT;
```

However, if the preceding program fragment is rewritten to include the assignment to J, shown in the fragment as a comment, the array address in the IF statement must be partially recalculated.

This optimization is called common subexpression elimination. The optimizer also reorders expressions so that the most complicated parts are done first, when more registers for temporary values are available. It also does several other optimizations, such as:

1. constant folding not done by the front end

2. strength reduction (changing multiplications and divisions into shifts when possible)

3. peephole optimization (removing additions of zero, multiplications by one, and changing A := A + 1 to an internal increment memory Icode)

The optimizer works by building a tree out of the intermediate codes for each statement and then transforming the list of statement trees.

There are seven internal passes per basic block:

1. statement tree construction from the Icode stream
2. preliminary transformations to set address/value flags
3. length checks and type coercions
4. constant and address folding, and expression reordering
5. peephole optimization and strength reduction
6. machine-dependent transformations
7. common subexpression elimination

Finally, the optimizer calls the code generator to translate the basic block from tree form to machine code. The code generator must translate these trees into actual machine code. It uses a series of templates to generate more efficient code for special cases. For example, there is a series of templates for the addition operator. The first template checks for an addition of the constant one. If this addition is found, the template generates an increment instruction.

If the template does not find an addition of one, then it gives up, and the next template gets control and checks for an addition of any constant. If this is found, the second template generates an add immediate instruction.

The final template in the series must handle the general case. It moves the operands into registers (by recursively calling the code generator itself), then generates an add register instruction. There is a series of templates for every operation. The code generator must also keep track of register contents, and several memory segment addresses (code, static variables, constant data, etc.). The code generator must also allocate any needed temporary variables.

The code generator writes a file of binary intermediate code (BINCOD), which contains actual byte values for machine instructions, symbolic references to external routines and variables, and other kinds of data. A final internal pass reads the BINCOD file and writes the object code file.

Pass Three

This short pass reads both the BINCOD file, described in the previous section, and a version of the symbol table file as updated by the optimizer and code generator. Using the data in these files, it writes the generated code in an assembler-like format.

APPENDIX C
RUNTIME ARCHITECTURE

CONTENTS

A successful Pascal compilation produces an "object" file that can be linked with other files to produce an executable file. Object files can come from any of the following:

1. Pascal programs, modules or units
2. User code in other high level languages
3. Assembly language
4. Routines in standard runtime modules that support facilities such as error handling, heap variable allocation, or input/output

RUNTIME ROUTINES

Pascal runtime entry points and variables conform to the naming convention: all names are six characters, and the last three are a unit identification letter followed by the letters "QQ". The following show the unit identifier suffixes:

Suffix	Unit Function
AQQ	Complex real
BQQ	Compile time utilities
CQQ	Encode, decode
DQQ	Double precision real
EQQ	Error handling
FQQ	Pascal file system
GQQ	Generated code helpers
HQQ	Heap allocator
IQQ	Generated code helpers
JQQ	Generated code helpers
KQQ	FCB definition
LQQ	STRING, LSTRING
MQQ	Reserved
NQQ	Long integer
OQQ	Other miscellaneous routines
PQQ	Reserved
QQQ	Reserved
RQQ	Real (single precision)
SQQ	Set operations
TQQ	Reserved
UQQ	Operating system file system
WQQ	Reserved
XQQ	Initialize/terminate
YQQ	Special utilities
ZQQ	Reserved

MEMORY ORGANIZATION

The memory in the B 20 is divided into segments, each containing up to 64K bytes. The relocatable object format and linker also put segments into classes and groups. All segments with the same class name are loaded next to each other. All segments with the same group name must reside in one area up to 64K long; that is, all segments in a group can be accessed with one B 20 segment register.

Pascal defines a single group, named DGROUP, which is addressed using the DS or SS segment register. Normally, DS and SS contain the same value, although DS may be changed temporarily to some other segment and changed back again. SS is never changed; its segment registers always contain abstract "segment values" and the contents are never examined or operated on. Long addresses, such as ADS variables use the ES segment register for addressing.

Memory is allocated within DGROUP for all static variables, constants which reside in memory, the stack, and the heap. Memory in DGROUP is allocated from the top down; that is, the highest addressed byte has DGROUP offset 65535, and the lowest allocated byte has some positive offset. This allocation means offset zero in DGROUP may address a byte in the code portion of memory, in the operating system below the code, or even below absolute memory address zero (in the latter case the values in DS and SS are "negative").

DGROUP has two parts:

1. a variable length lower portion containing the heap and the stack

2. a fixed length upper portion containing static variables and constants.

After your program is loaded, during initialization (in ENTX6L), the fixed upper portion is moved upward as much as possible to make room for the lower portion. If there is enough memory, DGROUP is expanded to the full 64K bytes; if there is not enough for this, it is expanded as much as possible.

1. 0000:0000

   The beginning of memory on the B 20 system contains interrupt vectors, which are segmented addresses. Usually the first 32 to 64 are reserved for the operating system. Following these vectors is the resident portion of BTOS.

   BTOS provides for loading additional code above it, which remains resident and is considered part of the operating system as well. Examples of resident additional code are for a print spooler, queue manager, etc.

2. BASE:0000

   Here, BASE means the starting location for loaded programs, sometimes called the transient program area. When you invoke a Pascal program, loading begins here. The beginning of your program contains the code portion, with one or more code segments. These code segments are in the same order as the object modules given to the linker, followed by object modules loaded from libraries.

3. DGROUP:LO

   Next comes the DGROUP data area, containing the following:

Segment	Class	Description
HEAP	MEMORY	Pointer variables, some files
MEMORY	MEMORY	(not used)
STACK	STACK	Frame variables and data
DATA	DATA	Static variables
CONST	CONST	Constant data

   The stack and the heap grow toward each other, the stack downward and the heap upward.

4. DGROUP:TOP

   Here TOP means 64K bytes (4K paragraphs) above DGROUP:0000 (i.e., just past the end of DGROUP).

5. HIMEM:0000

   The segment named HIMEM (class HIMEM) gives the highest used location in the program. The segment itself contains no data, but its address is used during initialization. Available memory starts here and can be accessed with ADS variables.

INITIALIZATION AND TERMINATION

Every executable file contains one, and only one, starting address. As a rule, when Pascal object modules are involved, this starting address is at the entry point BEGXQQ in the module PASSMAX. A Pascal program (as opposed to a module or implementation) has a starting address at the entry point ENTGQQ. BEGXQQ calls ENTGQQ.

The following discussion assumes that an Pascal main program along with other object modules is loaded and executed. However, you can also link a main program in assembly or some other language with other object modules in Pascal. In this case, some of the initialization and termination done by the ENTX module may need to be done elsewhere.

When a program is linked with the runtime library and execution begins, several levels of initialization are required. The levels, in the order in which they occur, are the following:

1. machine-oriented initialization
2. runtime initialization
3. program and unit initialization

Machine Level Initialization

The entry point of a Pascal load module is the routine BEGXQQ, in the module PASSMAX. BEGXQQ does the following:

1. It moves constant and static variables upward, creating a gap for the stack and the heap. It sets the stack pointer to the top of this area. The initial stack pointer is put into PUBLIC variable STKBQQ and is used to restore the stack pointer after an interprocedure GOTO to the main program.
2. It sets the frame pointer to zero.
3. It initializes a number of PUBLIC variables to zero or NIL. These include:

   RESEQQ, machine error context
   CSXEQQ, source error context list header
   PNUXQQ, initialized unit list header
   HDRFQQ, Pascal open file list header

4. It sets machine-dependent registers, flags, and other values.
5. It sets the heap control variables. BEGHQQ and CURHQQ are set to the lowest address for the heap: the word at this address is set to a heap block header for a free block the length of the initial heap. ENDHQQ is set to the address of the first word after the heap. The stack and the heap grow together, and the PUBLIC variable STKHQQ is set to the lowest legal stack address (ENDHQQ, plus a safety gap).

7. It calls INIUQQ, the file unit initializer. If the file unit is not used and you don't want it loaded, a dummy INIUQQ routine that just returns must be loaded.

8. It calls BEGOQQ, the escape initializer. In a normal load module, an empty BEGOQQ that only returns is included. However, this call provides an escape mechanism for any other initialization. For example, it could initialize tables for an interrupt driven profiler or a runtime debugger.

9. It calls ENTGQQ, the entry point of your Pascal program.

Program Level Initialization

Your main program continues the initialization process and INIFQQ which is a parameterless procedure is called first. If the main program is in Pascal, and FORTRAN file routines will be used, you must call INIVQQ to initialize the FORTRAN file system.

Pascal main programs automatically call INIFQQ. To avoid loading the file system, you must provide an empty procedure to satisfy one or both of these calls.

After file initialization, ENTEQQ is called to set the source error context (but only if $ENTRY is on during compilation). Next, each file at the program level gets an initialization call to NEWFQQ.

After static data initialization comes unit initialization. Every USES clause in the source, including those in INTERFACEs, generates a call to the initialization code for the unit.

Units may or may not contain initialization code. If the interface contains a trailing pair of BEGIN and END statements, then initialization code in the implementation is presumed. Units are initialized in the order that the USES clauses are encountered.

Finally, any program parameters are read or otherwise initialized, and your program begins. In general, except for INPUT and OUTPUT, PPMFQQ is called for each parameter to set the parameter's string value as the next line in the file INPUT. Then one of the READFN routines "reads" and decodes the value, assigning it to the parameter. The parameter's identifier is passed to PPMFQQ for use as a prompt. PPMFQQ first calls PPMUQQ to get the text of any command line parameter or other parameters. If PPMUQQ returns an error, then PPMFQQ does the prompting and reads the response directly.

User unit initialization is much like user program initialization. The following actions occur:

1. error context initialization if $ENTRY metacommand was on during compilation

2. variable (file) initialization

3. unit initialization for USES clause

4. user's unit initialization

Calls to initialize a unit may come from more than one unit. The unit interface has a version number, and each initialization call must check that the version number in effect when the unit was used in another compilation is the same as the version number in effect when the unit implementation itself was compiled. Except for this, unit initialization calls after the first one should have no effect; i.e., a unit's initialization code should be executed only once. Both version-number checking and single, initial-code execution are handled with code automatically generated at the start of the body of the unit. This has the effect of:

```
IF INUXQQ (useversion, ownversion, initrec, unitid) THEN
RETURN
```

The interface version number used by the compiland using the interface is always passed as a value parameter to the implementation initialization code. This is passed as "useversion" to INUXQQ. The interface version number in the implementation itself is passed as "ownversion" to INUXQQ. INUXQQ generates an error if the two are unequal.

INUXQQ also maintains a list of initialized units. INUXQQ returns true if the unit is found in the list, or else puts the unit in the list and returns false. The list header is PNUXQQ. A list entry passed to INUXQQ as "initrec" is initialized to the address of the unit's identifier (unitid), plus a pointer to the next entry.

User modules (and uninitialized implementations of units) may have initialization code, much like a program and unit implementation's initialization code, but without user initialization code or INUXQQ calls.

The initialization call for a module or uninitialized unit cannot be issued automatically. When the module is compiled, a warning is given if an initialization call will be required (i.e., if there are any files declared or USES clauses). To initialize a module, declare the module name as an external procedure and call it at the beginning of the program.

Program Termination

Program termination occurs in one of three ways:

1. The program may terminate normally, in which case the main program returns to BEGXQQ, at the location named ENDXQQ.

2. The program may abort because of an error condition, either with a user call to ABORT or a runtime call to an error handling routine. In either case, an error message, error code, and error status are passed to EMSEQQ, which does whatever error handling it can and calls ENDXQQ.

3. ENDXQQ can be declared in an external procedure and called directly.

ENDXQQ first calls ENDOQQ, the escape terminator, which normally just returns to ENDXQQ. Then ENDXQQ calls ENDYQQ, the generic file system terminator. ENDYQQ closes all open Pascal files, using the file list headers HDRFQQ and HDRVQQ. ENDXQQ calls ENDUQQ, the file unit terminator. Finally, ENDXQQ calls ENDX87 to terminate the real number processor (emulator). As with INIUQQ, INIFQQ, if your program requires no file handling, you will need to declare empty parameterless procedures for ENDYQQ and ENDUQQ. The main initialization and termination routines are in module ENTX. Procedures for BEGOQQ and ENDOQQ are in modules MISOALT1 and MISOALT2. ENDYQQ is in module MISY.

ERROR HANDLING

Runtime errors are detected in one of four ways:

1. The user program calls EMSEQQ (i.e., ABORT).

2. A runtime routine calls EMSEQQ.

3. An error checking routine in the error module calls EMSEQQ.

4. An internal helper routine calls an error message routine in the error unit that, in turn, calls EMSEQQ.

Handling an error detected at runtime usually involves identifying the type and location of the error and then terminating the program. The error type has three components:

1. a message

2. an error number

3. an error status

The error message describes the error, and the number can be used to look up for more information (see Appendix D, "Error Messages," of this Manual). The error status value is undefined, although for file system errors it may be an operating system return code. However, the error status value may also be used for other special purposes.

The following is a classification of the error codes:

Range	Classification
1- 999	Reserved for user ABORT calls
1000-1099	Unit U file system errors
1100-1199	Unit F file system errors
1300-1999	Reserved
2000-2049	Heap, stack, memory
2050-2099	Ordinal and long integer arithmetic
2100-2149	Real and double real arithmetic
2150-2199	Structures, sets and strings
2200-2499	Reserved
2450-2499	Other internal errors
2500-2999	Reserved

An error location has two parts:

1. the machine error context
2. the source program context

The machine error context is the program counter, stack pointer, and frame pointer at the point of the error. The program counter is always the address following a call to a runtime routine (e.g., a return address). The source program context is optional; it is controlled by metacommands. If the $ENTRY metacommand is on, the program context consists of:

1. the source filename of the compiland containing the error
2. the name of the routine in which the error occurred (program, unit, module, procedure, or function)
3. the line number of the routine in the listing file
4. the page number of the routine in the listing file

If the $LINE metacommand is also on, the line number of the statement containing the error is also given. Setting $LINE also sets $ENTRY.

Machine Error Context

Runtime routines are compiled by default with the $RUNTIME metacommand set. This causes special calls to be generated at the entry and exit points of each runtime routine. The entry call saves the context at the point where a runtime routine is called by the user program. This context consists of the frame pointer, stack pointer, and program counter. As a consequence of this saving of context, if an error occurs in a runtime routine, the error location is always in the user program. This is true even if runtime routines call other runtime routines. The exit call that is generated restores the context. The runtime entry helper, BRTEQQ, uses the runtime values as shown below.

Value	Description
RESEQQ	Stack pointer
REFEQQ	Frame pointer
REPEQQ	Program counter offset
RECEQQ	Program counter segment

The first thing that BRTEQQ does is examine RESEQQ. If this value is not zero, the current runtime routine was called from another runtime routine and the error context has already been set, so it just returns. If RESEQQ is zero, however, the error context must be saved. The caller's stack pointer is determined from the current frame pointer and stored in RESEQQ. The address of the caller's saved frame pointer and return address (program counter) in the frame is determined. Then the caller's frame pointer is saved in REFEQQ. The caller's program counter (i.e., BRTEQQ's caller's return address) is saved: the offset in REPEQQ and the segment (if any) in RECEQQ.

The runtime exit helper, ERTEQQ, has no parameters. It determines the caller's stack pointer (again, from the frame pointer) and compares it against RESEQQ. If these values are equal, the original runtime routine called by your program is returning, so RESEQQ is set back to zero.

EMSEQQ uses RESEQQ, REFEQQ, REPEQQ, and RECEQQ to display the machine error context.

Source Error Context

Giving the source error context involves extra overhead, since source location data must be included in the object code in some form. This is done with calls which set the current source context as it occurs. These calls can also be used to break program execution as part of the debug process. The overhead of source location data, especially line number calls, can be significant. Routine entry and exit calls, while requiring more overhead, are much less frequent, so the overhead is less.

The procedure entry call to ENTEQQ passes two VAR parameters: the first is an LSTRING containing the source filename; the second is a record that contains the following:

1. the line number of the procedure (a WORD)

2. the page number of the procedure (a WORD)

3. the procedure or function identifier (an LSTRING)

The filename is that of the compiland source (e.g., the main source filename, not the names of any $INCLUDE files). The procedure identifier is the full identifier used in the source, not the linker name. If one name is given in an INTERFACE and another in a USES clause, the USES identifier is used. The line and page are those designated by the procedure header.

Entry and exit calls are also generated for the main program, unit initialization, and module initialization, in which case the identifier is the program, unit, or module.

The procedure exit call to EXTEQQ does not pass any parameters. It pops the current source routine context off a stack maintained in the heap.

The line number call to LNTEQQ passes a line number as a value parameter. The current line number is kept in the PUBLIC variable CLNEQQ. Since the current routine is always available (because $LINE implies $ENTRY), the compiland source filename and routine containing the line are available along with the line number. Line number calls are generated just before the code in the first statement on a source line. The statement can, of course, be part of a larger statement. The $LINE+ metacommand should be placed at least a couple of symbols before the start of the first statement intended for a line number call ($LINE- also takes effect "early").

Most of the error handling routines are in modules ERREALT and PASE. The source error context entry points ENTEQQ, EXTEQQ, and LNTEQQ are in the debug module, DEBE.

APPENDIX D
ERROR MESSAGES

CONTENTS

This appendix lists all of the error numbers and messages you are likely to encounter while using the Pascal Compiler and runtime system. These error conditions fall into the following categories:

1. compile time warnings
2. compile time errors caught
3. compiler internal errors
4. errors (both compile time and runtime) defined by the ISO standard not caught in this extended Pascal
5. runtime file system errors
6. runtime nonfile system errors caught only if the appropriate switch is on
7. runtime nonfile system errors always caught

Error conditions may:

1. go undetected
2. be detected by the compiler
3. be detected by the runtime system

An error is "caught" if the compiler or runtime system detects the error and gives you a message. A "warning" is an error that is caught by the compiler but fixed so that the compiled source might run correctly. Substitution mistakes (e.g., using a colon (:) instead of an equal sign (=)) and some other syntax errors (e.g., using a semicolon (;) before an ELSE) are common errors that generate only a warning message and are fixed by the compiler. You should, however, go back into the source file and make corrections, or you will keep getting the same warning message every time you compile.

Compile time errors include all of the conditions described in this manual as "invalid," "illegal," "not permitted," and so on. The ISO standard defines a number of error conditions that are described as "errors not caught" in this Pascal. Generally, these are infrequent or very hard to detect conditions,and not caught as errors in this Pascal.

COMPILER FRONT END ERRORS

Front end error and warning messages consist of a number and a message. Most messages appear with a row of dashes and an arrow that points to the location of the error; three (messages 128,

129, and 13Ø) appear only after the body of the routine in which they occur. The word "Warning" identifies warnings as such; all other messages report errors in the program.

The front end recovers from most errors; that is, it corrects the condition and continues the compilation. There are, however, a few front end errors ("panic" errors) from which the compiler cannot recover. In these cases, you see the message:

```
Compiler Cannot Continue!
```

The compiler then does little else except list the rest of the program. These errors occur under the following circumstances:

1. There are more errors than the number n set by the $ERRORS metacommand.

2. An end of file occurs when not expected.

3. Identifier scopes are nested too deeply.

4. The compiler cannot find the keyword PROGRAM, MODULE, or IMPLEMENTATION.

5. The compiler cannot find the PROGRAM, MODULE, or IMPLEMENTATION identifier.

6. A file system error occurs. The message will include the filename and one of the following phrases:

```
HARD DATA      (check sum error)
DISK FULL      (disk is full)
FILE ACCESS    (file not found)
FILE SYSTEM    (other or internal error)
```

The front end may also get one of two compiler runtime errors:

1. Error: Compiler Out Of Memory

   This usually occurs when too many identifiers have been declared. See "Compiling and Linking Large Programs" under Chapter 14, "Compiling, Linking, Executing Programs", of this Manual for suggestions on how to handle this situation.

2. Error: Compiler Internal Error

   No matter what source program is compiled, this message should not appear. If it does, please report the condition to Burroughs Corporation.

If the word "Warning" appears before a message, the intermediate code files produced by the front end is correct. The condition that produced it is not severe, but is considered unsafe. Messages that indicate true errors halt any writing to intermediate files, which are discarded when the front end is finished.

The error message "Compiler" signifies the failure of an internal consistency check. No matter what source program is compiled, this message should not appear. If it does, please report the condition to Burroughs Corporation.

The following list of compiler front end errors includes the error number and message, with a brief explanation of the condition that generates the message.

Code	Message
101	Invalid Line Number
	There are two many lines in the source file (limit is 32767).
102	Line Too Long Truncated
	There are too many characters in the line (current limit is 142 characters).
103	Identifier Too Long Truncated
	An identifier is longer than the maximum length permitted and has been truncated.
104	Number Too Long Truncated
	A numeric constant is too long and has been truncated. Numeric constants are limited to the same maximum length as identifiers.
105	End Of String Not Found
	The line ended before the closing quotation mark was found.
106	Assumed String

The compiler encountered double quotation marks (") or back-quotes (`) and assumed that they enclose a string. Use single quotation marks instead.

107 Unexpected End Of File

While scanning, the compiler found an unexpected end-of-file in a number, metacommand, or other illegal location.

108 Metacommand Expected Command Ignored

The compiler found a dollar sign ($) at the start of a comment, but not a metacommand identifier.

109 Unknown Metacommand Ignored

The compiler found a metacommand identifier that it didn't recognize or that is invalid.

110 Constant Identifier Unknown Or Invalid Assumed Zero

The constant identifier following a metacommand is unknown (as in $DEBUG: A) or not a constant of the right type. The compiler has replaced the unknown or incorrect value with zero.

111 [Unassigned]

112 Invalid Numeric Constant Assumed Zero

The constant following a metacommand was a numeric constant (e.g., $DEBUG: 123456) that has the wrong format or is out of range. The compiler has replaced the incorrect value with zero.

113 Invalid Meta Value Assumed Zero

The value following a metacommand is neither a constant nor an identifier. The compiler has replaced the incorrect value with zero.

114 Invalid Metacommand

The compiler expected but did not find one of the following after a metacommand: +, -, or :. The metacommand has been ignored by the compiler.

115 Wrong Type Value For Metacommand Skipped

The value following the metacommand was an integer, but should have been a string (or vice versa). The metacommand has been ignored by the compiler.

116 Meta Value Out Of Range Skipped

The integer value given for the $LINESIZE metacommand was below 16 or above 160. Or, "n" is neither 4 nor 8 for $REAL:n, nor 2 for $INTEGER. In any of these cases, the compiler ignores the metacommand.

117 File Identifier Too Long Skipped

The string value given for the filename in a $INCLUDE metacommand was too long. The metacommand has been ignored. The maximum is 96 characters.

118 Too Many File Levels

There are too many nested levels of files brought in by the $INCLUDE metacommand. The $INCLUDE metacommand is ignored.

119 Invalid Initialize Metacommand

A $POP metacommand has no corresponding $PUSH metacommand.

120 CONST Identifier Expected

The compiler didn't find an identifier following an $INCONST metacommand. The $INCONST metacommand is ignored.

121 Invalid INPUT Number Assumed Zero

The user input invoked by $INCONST was invalid in some way and is assumed to be zero.

122 Invalid Metacommand Skipped

The compiler found an $IF metacommand but no subsequent $THEN or $ELSE. The $IF command has been

ignored.

123 Unexpected Metacommand Skipped

The compiler found a $THEN metacommand unrelated to any $IF metacommand. The $THEN command is ignored.

124 Unexpected Metacommand

The compiler found a metacommand not enclosed in comment delimiters, but processed it anyway.

126 Invalid Real Constant

The compiler found a type REAL constant with a leading or a trailing decimal point. The constant's value is accepted anyway.

127 Invalid Character Skipped

The compiler found a character in the source file that is not acceptable in program text.

128 Forward Proc Missing: <procedure>

The compiler found a procedure or function declared FORWARD but couldn't find the procedure or function itself. This message appears in the symbol table area of the listing file.

129 Label Not Encountered: <label>

The compiler couldn't find any use of a label you declared in a LABEL section. This message occurs in the symbol table area of the listing file.

130 Program Parameter Bad: <parameter>

The compiler encountered this program parameter, which was never declared or has an unacceptable type. This message occurs in the symbol table area of the listing file.

131 [Unassigned]

132 [Unassigned]

133 Type Size Overflow

The data type declared implies a structure bigger than the maximum of 65534 bytes.

134 Constant Memory Overflow

Constant memory allocation has exceeded the maximum of 65534 bytes.

135 Static Memory Overflow

Static memory allocation has exceeded the maximum of 65534 bytes.

136 Stack Memory Overflow

Stack frame memory allocation has exceeded the maximum of 65534 bytes.

137 Integer Constant Overflow

The value of a type INTEGER, signed constant expression is out of range.

138 Word Constant Overflow

The value of a type WORD or other unsigned constant expression is out of range.

139 Value Not In Range For Record

In a structured constant, long form of the NEW, DISPOSE, or SIZEOF procedure, or other application, the record tag value is not in the range of the variant.

140 Too Many Compiler Labels

The compiler needs internal labels, and the program is too big. You must break your program into smaller pieces.

141 Compiler

142 Too Many Identifier Levels

The identifier scope level exceeds 15. This is a panic error!

143 Compiler

144 Compiler

This error may occur if the PASKEY file format is incorrect.

145 Identifier Already Declared

The compiler found an identifier declared more than once in a given scope level.

146 Unexpected End Of File

While parsing, the compiler found an end-of-file where it shouldn't be in a statement, declaration etc.

147 : Assumed =

The compiler found a colon where there should have been an equal sign and proceeded as if the correct symbol were present.

148 = Assumed :

The compiler found an equal sign where it expected a colon and proceeded as if the correct symbol were present.

149 := Assumed =

The compiler found a colon followed by an equal sign where it expected an equal sign only and proceeded as if the correct symbol were present.

I5Ø = Assumed :=

The compiler found an equal sign where it expected a colon following by an equal sign and proceeded as if

the correct symbol were present.

151 [Assumed (

The compiler found a left bracket where it expected a left parenthesis and proceeded as if the correct symbol were present.

152 (Assumed [

The compiler found a left parenthesis where it expected a left bracket and proceeded as if the correct symbol were present.

153) Assumed]

The compiler found a right parenthesis where it expected a right bracket and proceeded as if the correct symbol were present.

154] Assumed)

The compiler found a right bracket where it expected a right parenthesis and proceeded as if the correct symbol were present.

155 ; Assumed ,

The compiler found a semicolon where it expected a comma and proceeded as if the correct symbol were present.

156 , Assumed ;

The compiler found a comma where it expected a semicolon and proceeded as if the correct symbol were present.

157..161 [Unassigned]

162 Insert Symbol

The compiler didn't find a symbol it expected, but proceeded as if it were present. This message should not occur; it is a minor compiler error. If it does, please report it to Burroughs Corporation.

163 Insert ,

The compiler didn't find a comma where it expected one, but proceeded as if it were present.

164 Insert ;

The compiler didn't find a semicolon where it expected one, but proceeded as if it were present.

165 Insert =

The compiler didn't find an equal sign where it expected one, but proceeded as if it were present.

166 Insert :=

The compiler didn't find a colon followed by an equal sign where it expected one, but proceeded as if it were present.

167 Insert OF

The compiler didn't find an OF where it expected one, but proceeded as if it were present.

168 Insert]

The compiler didn't find a right bracket where it expected one, but proceeded as if it were present.

169 Insert)

The compiler didn't find a right parenthesis where it expected one, but proceeded as if it were present.

170 Insert [

The compiler didn't find a left bracket where it expected one, but proceeded as if it were present.

171 Insert (

The compiler didn't find a left parenthesis where it expected one, but proceeded as if it were present.

172 Insert DO

The compiler didn't find a DO where it expected one, but proceeded as if it were present.

173 Insert :

The compiler didn't find a colon where it expected one, but proceeded as if it were present.

174 Insert .

The compiler didn't find a period where it expected one, but proceeded as if it were present.

175 Insert ..

The compiler didn't find a double period where it expected one, but proceeded as if it were present.

176 Insert END

The compiler didn't find an END where it expected one, but proceeded as if it were present.

177 Insert TO

The compiler didn't find a TO where it expected one, but proceeded as if it were present.

178 Insert THEN

The compiler didn't find a THEN where it expected one, but proceeded as if it were present.

179 Insert *

The compiler didn't find an asterisk where it expected one, but proceeded as if it were present.

180..184 [Unassigned]

185 Invalid Symbol Begin Skip

The compiler found a symbol it expected, but only after some other invalid symbols. The invalid symbols were skipped, beginning at the point where message 185 appears and ending where message 186 appears.

186 End Skip

The compiler found a symbol it expected, but only after some other invalid symbols. The invalid symbols were skipped, beginning at the point where message 185 appears and ending where message 186 appears.

187 End Skip

This message marks the end of skipped source text for any message, except message 185, that ended with the phrase "Begin Skip".

188 Section Or Expression Too Long

The compiler has reached its limit. Try rearranging the program or breaking up an expression with assignments to intermediate values.

189 Invalid Set Operator Or Function

Your source file includes an incorrect use of a set operator or function (for example, MOD operator or ODD function with sets).

190 Invalid Real Operator Or Function

Your source file includes an incorrect use of an operator or function on a REAL value (for example, MOD operator or ODD function with reals).

191 Invalid Value Type For Operator Or Function

For example, MOD operator or ODD function with enumerated type.

192..194 [Unassigned]

195 Compiler

196 Zero Size Value

Your source file includes the empty record "RECORD END" as if it had a size.

197 Compiler

198 Constant Expression Value Out Of Range

The value of a constant expression is out of range in an array index, subrange assignment, or other subrange.

199 Integer Type Not Compatible With Word Type

An expression tries to mix INTEGER and WORD type values. This common error indicates confusing signed and unsigned arithmetic; either change the positive signed value to unsigned with WRD () or change the unsigned value (< MAXINT) to signed with ORD ().

200 [Unassigned]

201 Types Not Assignment Compatible

You have attempted to use incompatible types in an assignment statement or value parameter. See "Type Compatibility," under Chapter 6, Data Types, for type compatibility rules.

202 Types Not Compatible In Expression

You have attempted to mix incompatible types in an expression. See "Type Compatibility," under Chapter 6, Data Types for type compatibility rules.

203 Not Array Begin Skip

A variable followed by a left bracket (or parenthesis) is not array. The compiler has skipped from here to where message 187 appears.

204 Invalid Ordinal Expression Assumed Integer Zero

The expression has the wrong type or a type that is not ordinal. The compiler assumes the value of the expression to be zero.

205 Invalid Use Of PACKED Components

A component of a PACKED structure has no address (it may not be on a byte boundary) and cannot be passed by reference.

206 Not Record Field Ignored

A variable followed by a period is not a record, address, or file, and has been ignored by the compiler.

207 Invalid Field

A valid field name does not follow a record variable and a period, and has been ignored by the compiler.

208 File Dereference Considered Harmful

When the compiler calculates the address of a file buffer variable, it cannot do the special actions normally done with buffer variables (i.e., lazy evaluation, for textfiles). Since the buffer variable at this address may not be valid, such a practice is considered harmful.

209 Cannot Dereference Value

The variable followed by an arrow is not a pointer, address, or file; therefore the compiler cannot dereference the value pointed to.

210 Invalid Segment Address

A variable resides at segmented address, but a default segment address is needed. You may need to make a local copy of the variable.

211 Ordinal Expression Invalid Or Not Constant

The compiler found an invalid or nonconstant expression where it expected a constant ordinal expression.

212..213 [Unassigned]

214 Out Of Range For Set 255 Assumed

The compiler found an element of a set constant whose ordinal value exceeded 255 and assumed a value of 255.

215 Type Too Long Or Contains File Begin Skip

The compiler found a structured constant that exceeded 255 bytes or either is or contains a FILE or LSTRING type.

216 Extra Array Components Ignored

The compiler found an array constant that had too many components for the array type. The excess components were ignored.

217 Extra Record Components Ignored

The compiler found a record constant that had too many components for the record type. The excess components were ignored.

218 Constant Value Expected Zero Assumed

The compiler found a nonconstant value in a structured constant and assumed its value was zero.

219 [Unassigned]

220 Compiler

221 Components Expected For Type

The compiler found too few components for the type of a structured constant.

222 Overflow 255 Components In String Constant

The compiler found a string constant that exceeded 255 bytes.

223 Use NULL

Use the predeclared constant NULL instead of two quotation marks.

224 Cannot Assign With Supertype Lstring

A super array LSTRING cannot be the source or the target of an assignment.

225 String Expression Not Constant

String concatenation with the asterisk applies only to constants.

226 String Expected Character 255 Assumed

The compiler found a string constant with no characters, perhaps the result of using NULL, and assumed the value CHR(255).

227 Invalid Address Of Function

An assignment or other address reference to the function value is not within the scope of the function. Or, RESULT is used outside the scope of the function.

228 Cannot Assign To Variable

Assignment to READONLY, CONST, or FOR control variable is not permitted.

229 [Unassigned]

230 Unknown Identifier Assumed Integer Begin Skip

The compiler found an unknown identifier, for which it requires an address, and has skipped to a comma, semicolon, or right parenthesis.

231 VAR Parameter Or WITH Record Assumed Integer Begin Skip

The compiler found an invalid symbol where it requires an address, and has skipped to a comma, semicolon, or right parenthesis.

232 Cannot Assign To Type

The target of an assignment is a file or cannot be assigned for some other reason.

233 Invalid Procedure Or Function Parameter Begin Skip

The compiler found an incorrect use of an intrinsic procedure or function. The error could be one of the following:

1. The first parameter of NEW or DISPOSE is not a pointer variable.
2. The record tag value of a NEW, DISPOSE, or SIZEOF procedure couldn't be found.
3. The super array in a NEW, DISPOSE, or SIZEOF procedure had too many bounds.
4. The super array in a NEW, DISPOSE, or SIZEOF procedure had too few bounds.
5. The super array for a NEW or SIZEOF procedure has been given no bounds.
6. You attempted to use a WRD or ORD function on a value not of an ordinal type.
7. You attempted to use the LOWER or UPPER functions on an invalid value or type.
8. PACK or UNPACK on super array or file, or an array that is or is not packed as expected.
9. The first parameter for a RETYPE is not a type identifier.
10. The parameter for a RESULT function is not a function identifier.
11. You attempted to use an intrinsic procedure or function that was not available.
12. The ORD or WRD of an INTEGER4 value is out of range.
13. The parameter given for HIWORD or LOWORD is not an ordinal or INTEGER4.

234 Type Invalid Assumed Integer

The parameter given to READ, WRITE, ENCODE, or DECODE is not of type INTEGER, WORD, INTEGER4, REAL, BOOLEAN, enumerated, a pointer; or, the parameter given for a READ or WRITE is not of type CHAR, STRING, LSTRING; or, the parameter for a READFN is

not of one of these types or type FILE. The compiler has assumed it to be of type INTEGER. This error also occurs if a program parameter does not have a readable type, in which case the error occurs at the keyword BEGIN for the main program.

235 Assumed File INPUT

Because the first parameter for a READFN is not a file, INPUT is assumed.

236 Invalid Segment For File

File parameters must always reside in the default segment.

237 Assumed INPUT

INPUT was not given as a program parameter and has been assumed.

238 Assumed OUTPUT

OUTPUT was not given as a program parameter and has been assumed.

239 Not Lstring Or Invalid Segment

The target of a READSET, ENCODE, or DECODE must be an LSTRING in the default segment. One or both of these conditions is missing.

240 [Unassigned]

242 File Parameter Expected Begin Skip

The READSET procedure expects, but cannot find, a textfile parameter. The compiler has ignored the procedure and resumed where message 187 appears.

243 Character Set Expected

The READSET procedure expects, but cannot find, a SET OF CHAR parameter.

244 Unexpected Parameter Begin Skip

The compiler found more than one parameter given for an EOF, EOLN, or PAGE, and has ignored the extra.

245 Not Text File

You attempted to use an EOLN, PAGE, READLN, or WRITELN on some file other than a textfile.

246..247 [Unassigned]

248 Size Not Identical

The RETYPE function may not work as intended, since the parameters given are of unequal length.

249 Procedural Type Parameter List Not Compatible

The parameter lists for formal and actual procedural parameters are not compatible. That is, the number of parameters, the function result type, a parameter type, or attributes are different.

250 Cannot Use Procedure With Attribute

You attempted to call a procedure with an invalid attribute

251 Unexpected Parameter Begin Skip

The compiler found a left parenthesis, indicating a procedure or function, but no parameters, and has skipped to where message 187 appears.

252 Cannot Use Procedure Or Function As Parameter

You attempted to pass this intrinsic procedure or function as a parameter, which is not permitted.

253 Parameter Not Procedure Or Function Begin Skip

The compiler expected, but cannot find, a procedural parameter here, and has skipped to where message 187 appears.

254 Supertype Array Parameter Not Compatible

The actual parameter given is not of the same type or is not derived from the same super type as the formal parameter.

255 Compiler

256 VAR Or CONST Parameter Types Not Identical

The actual and formal reference parameter types are not identical, as they must be.

257 Parameter List Size Wrong Begin Skip

The compiler found too many or too few parameters in a list. If too many, the excess has been skipped.

258 Invalid Procedural Parameter To EXTERN

A procedure or function that is neither PUBLIC nor EXTERN is being passed as a parameter to a procedure or function declared EXTERN. (The compiler invokes the actual procedure or function with intrasegment calls, and so cannot pass them to an external code segment.)

259 Invalid Set Constant For Type

The set is not constant, base types are not identical, or the constant is too big.

260 Unknown Identifier In Expression Assumed Zero

The identifier in an expression is undefined or possibly misspelled.

261 Identifier Wrong In Expression Assumed Zero

The identifier in an expression is incorrect (e.g., file type identifier) and has been assumed to be zero.

262 Assumed Parameter Index Or Field Begin Skip

After error 260 or 261, anything in parentheses or square brackets, or a dot followed by an identifier, is skipped.

262..264 [Unassigned]

265 Invalid Numeric Constant Assumed Zero

There is a decode error in an assumed INTEGER or INTEGER4 literal constant; the number is too big, has invalid characters, etc. The incorrect constant has been assumed to be zero.

267 Invalid Real Numeric Constant

There is a decode error in an assumed type REAL literal constant; the number is too big, has invalid characters, etc.

268 Cannot Begin Expression Skipped

A symbol that cannot start an expression has been deleted.

269 Cannot Begin Expression Assumed Zero

A symbol that cannot start an expression has been prefixed with a zero.

270 Constant Overflow

The divisor in a DIV or MOD function is the constant zero (INTEGER or WORD), which is not permitted.

271 [Unassigned]

272 Word Constant Overflow

A WORD constant minus a WORD constant has given a negative result.

273..274 [Unassigned]

275 Invalid Range

The lower bound of a subrange is greater than the upper bound (e.g. 2..1).

276 CASE Constant Expected

The compiler expects, but cannot find, a constant value for a CASE statement or record variant.

277 Value Already In Use

In a CASE statement or record variant, the value has already been assigned (as in CASE 1..3: XXX; 2: YYY; END).

279 Label Expected

The compiler expects, but cannot find, a label.

280 Invalid Integer Label

A label uses nondecimal notation (e.g. 8#77), which is not allowed.

281 Label Assumed Declared

The compiler found a label that did not appear in the LABEL section.

282 [Unassigned]

283 Expression Not Boolean Type

The expression following an IF, WHILE, or UNTIL statement must be BOOLEAN.

284 Skip To End Of Statement

The compiler found, and has skipped, an unexpected ELSE or UNTIL clause.

285 Compiler

286 ; Ignored

The compiler found, and has ignored, a semicolon before an ELSE statement. (The semicolon is not required in this case.)

287 [Unassigned]

288 : Skipped

The compiler found, and has ignored, a colon after an OTHERWISE statement. (The colon is not required in this case.)

289 Variable Expected For FOR Statement Begin Skip

The compiler expects, but cannot find, a variable identifier after a FOR statement and has skipped to where message 187 appears.

290 [Unassigned]

291 FOR Variable Not Ordinal Or Static Or Declared In Procedure

The compiler has found an incorrect control variable in a FOR statement. Specifically, the control variable is, but should not be, one of the following:

1. type REAL, INTEGER4, or another non-ordinal type
2. the component of an array, record, or file type
3. the referent of a pointer type or address type
4. in the stack or heap, unless locally declared
5. nonlocally declared, unless in static memory
6. a reference parameter (VAR or VARS parameter)
7. a variable with a segmented ORIGIN attribute

292 Skip To :=

The compiler expects, but cannot find, an assignment in a FOR statement, and has skipped to the next :=.

293 GOTO Invalid

The GOTO or label here involves an invalid GOTO statement.

294 GOTO Considered Harmful

As directed, if the $GOTO metacommand is on, the compiler has found a GOTO statement.

296 Label Not Loop Label

The label after a BREAK or CYCLE statement is not a loop label (i.e., does not label a FOR, WHILE, or REPEAT statement).

297 Not In Loop

The compiler has found a BREAK or CYCLE statement outside a FOR, WHILE, or REPEAT statement.

298 Record Expected Begin Skip

The compiler expects, but cannot find, a record variable in a WITH statement and has skipped to where message 187 appears.

299 [Unassigned]

300 Label Already In Use Previous Use Ignored

The compiler found a label that has already appeared in front of a statement and has ignored the previous use.

301 Invalid Use Of Procedure Or Function Parameter

The compiler has found a procedure parameter used as a function or a function parameter used as a procedure.

302 [Unassigned]

303 Unknown Identifier Skip Statement

The compiler has found an undefined (or possibly misspelled) identifier at the beginning of a statement and has ignored the entire statement.

304 Invalid Identifier Skip Statement

The compiler has found an incorrect identifier at the beginning of a statement (e.g., file type identifier) and has ignored the entire statement.

305 Statement Not Expected

The compiler has found a MODULE or uninitialized IMPLEMENTATION with a body enclosed with the reserved words BEGIN and END.

306 Function Assignment Not Found

The compiler expects, but cannot find, an assignment of the value of a function somewhere in its body.

307 Unexpected END Skipped

The compiler found, and ignored, an END without a matching BEGIN, CASE, or RECORD.

308 Compiler

309 Attribute Invalid

The compiler found an attribute valid only for procedures and functions given to a variable, an attribute valid only for a variable given to a procedure or function, or an invalid mix of attributes (e.g., PUBLIC and EXTERN).

310 Attribute Expected

The compiler expects, but cannot find, a valid attribute, following the left bracket.

311 Skip To Identifier

The compiler skipped an invalid (i.e., unexpected) symbol to get to the identifier that follows.

312 Identifier Expected

The compiler found something not an identifier where it expected a list of identifiers.

313 [Unassigned]

314 Identifier Expected Skip To ;

The compiler expects, but cannot find, the declaration of a new identifier and has skipped to the next semicolon.

315 Type Unknown Or Invalid Assumed Integer Begin Skip

The return type for a parameter or function is incorrect; that is, it is not an identifier or is undeclared, or the value parameter or function value is a file or super array. The compiler has assumed the type is INTEGER and skipped to where message 187 appears.

316 Identifier Expected

The compiler expects, but cannot find, an identifier after the word PROCEDURE or FUNCTION in parameter list.

317 [Unassigned]

318 Compiler

319 Compiler

320 Previous Forward Skip Parameter List

The compiler found a definition of a FORWARD (or INTERFACE) procedure or function that unnecessarily repeats the parameter list and function return type.

321 Not EXTERN

The compiler found a procedure or function with the ORIGIN attribute but lacking the EXTERN attribute as well.

322 Invalid Attribute With Function Or Parameter

The compiler found an invalid attribute

323 Invalid Attribute In Procedure Or Function

The compiler has found a nested procedure or function that has attributes or is declared EXTERN. Neither of these conditions is permitted.

324 Compiler

325 Already Forward

You attempted to use FORWARD twice for the same procedure or function.

326 Identifier Expected For Procedure Or Function

The compiler expects, but cannot find, an identifier following the keywords PROCEDURE or FUNCTION.

327 Invalid Symbol Skipped

The compiler found, and ignored, a FORWARD or EXTERN directive in an interface.

328 EXTERN Invalid With Attribute

The compiler found an EXTERN procedure also declared PUBLIC. This is not permitted.

329 Ordinal Type Identifier Expected Integer Assumed Begin Skip

The compiler expects, but cannot find, an ordinal type identifier for a record tag type. It has skipped what is given in the source file and assumed type INTEGER.

330 Contains File Cannot Initialize

You have used a file in a record variant. This is allowed, but considered unsafe, and is not initialized automatically with the usual NEWFQQ call.

331 Type Identifier Expected Assumed Character

The compiler expects, but cannot find, an ordinal type identifier. It assumes that what it does find is of type CHAR.

332 [Unassigned]

333 Not Supertype Assumed String

The compiler has found what looks like a super array type designator. However, the type identifier is not for a super array type, so the compiler assumes it to be of the super array type STRING.

334 Type Expected Integer Assumed

The compiler expects, but cannot find, a type clause or type identifier and has assumed the expected type to be type INTEGER.

335 Out Of Range 255 For Lstring

The compiler has found an LSTRING designator whose upper bound exceeds 255.

336 Cannot Use Supertype Use Designator

A super array type can only be used as a reference parameter or a pointer referent. Other variables cannot be given a super array type. Use a super array designator.

337 Supertype Designator Not Found

The compiler expects, but cannot find, a super array designator that gives the upper bounds of the super array.

338 Contains File Cannot Initialize

The compiler has found a super array of a file type. While allowed, this is considered unsafe and is not initialized automatically with the usual NEWFQQ call.

339 Supertype Not Array Skip To; Assumed Integer

The compiler expects, but cannot find, the keyword ARRAY following SUPER in a type clause. It has assumed that the type is INTEGER and skipped to the next semicolon.

340 Invalid Set Range Integer Zero To 255 Assumed

The compiler has found an invalid range for the base type of a set and assumed it to be of type INTEGER with a range from zero to 255.

341 File Contains File

The compiler has found, but does not permit, a file type that contains a file type, either directly or indirectly.

342 PACKED Identifier Invalid Ignored

The compiler expects, but cannot find, one of words ARRAY, RECORD, SET, or FILE following the reserved word PACKED. Any type identifier following PACKED is not permitted.

343 Unexpected PACKED

The compiler found the keyword PACKED applied to one of the nonstructured types.

345 Skip To ;

The compiler expects, but cannot find, a semicolon at the end of a declaration (which is not at the end of the line). It has assumed the next semicolon is the end of the declaration.

346 Insert ;

The compiler expects, but cannot find, a semicolon at end of the declaration (which coincides with the end of a line). It has inserted a semicolon where it expected to find one.

347 Cannot Use Value Section With ROM Memory

If the $ROM metacommand is on, you may not also have

a VALUE section.

348 UNIT Procedure Or Function Invalid EXTERN

A required EXTERN declaration occurs later than it should in an IMPLEMENTATION. (Any interface procedures and functions not implemented must be declared EXTERN at the beginning.)

349 [Unassigned]

350 Not Array Begin Skip

The variable followed by a left bracket, in a VALUE section, is not an array.

351 Not Record Begin Skip

The variable followed by a period, in VALUE section, is not a record type.

352 Invalid Field

Within a VALUE section, the identifier assumed to be a field is not in the record.

353 Constant Value Expected

Within a VALUE section, a variable has been initialized to something other than a constant.

354 Not Assignment Operator Skip To ;

Within a VALUE section, the assignment operator is missing.

355 Cannot Initialize Identifier Skip To ;

Within a VALUE section, there is a symbol that is not a variable declared at this level in fixed (STATIC) memory. Or, it has an illegal ORIGIN or EXTERN attribute.

356 Cannot Use Value Section

A VALUE section has been incorrectly included in the INTERFACE, rather than in the IMPLEMENTATION.

357 Unknown Forward Pointer Type Assumed Integer

The identifier for the referent of a reference type declared earlier in this TYPE (or VAR) section was never declared itself.

358 Pointer Type Assumed Forward

The TYPE section includes a pointer or address type for which the referent type was already declared in an enclosing scope. Since the identifier for the referent type was declared again later in the same TYPE section, the compiler used the second definition. In the following example the forward type, REAL, is used:

```
PROGRAM OUTSIDE;
TYPE A = WORD;
PROCEDURE B;
TYPE C= ^A;
A = REAL;
```

359 Cannot Use Label Section

The compiler found a LABEL section incorrectly included in an INTERFACE, rather than in an IMPLEMENTATION.

360 Forward Pointer To Supertype

The referent of a reference type declared in this TYPE section is a super array type. The declaration the super array type doesn't occur until after the reference.

361 Constant Expression Expected Zero Assumed

An expression in a CONST section is not constant.

362 Attribute Invalid

A VAR section mixes incorrectly the PUBLIC or ORIGIN attribute with EXTERN. Or, ORIGIN appears in attribute brackets after the keyword VAR.

363 [Unassigned]

364 Contains File Initialize Module

The compiler found an uninitialized file variable in a module. You must call the module as a parameterless procedure to initialize the files.

365 Origin Variable Contains File Cannot Initialize

The compiler found an uninitialized file variable with the ORIGIN attribute. Since ORIGIN variables are never initialized, you must initialize this file yourself.

366 UNIT Identifier Expected Skip To ;

The compiler expects, but cannot find, an identifier after the keyword USES.

367 Initialize Module To Initialize UNIT

You must call the module as a procedure in order to initialize it (a USES clause triggers a unit initialization call).

368 Identifier List Too Long Extra Assumed Integer

In a USES clause with a list of identifiers, the compiler found more identifiers in the list than are constituents of the interface. The extra ones are assumed to be type identifiers identical to INTEGER.

369 End Of UNIT Identifier List Ignored

In a USES clause with a list of identifiers, the compiler found fewer identifiers in the list than are constituents of the interface. The remaining interface constituents are not provided as part of the USES clause.

370 [Unassigned]

371 UNIT Identifier Expected

An identifier is missing after the phrase "INTERFACE; UNIT".

372 Compiler

Compiler expects, but cannot find, the keyword UNIT in an INTERFACE.

373 Identifier In UNIT List Not Declared

One of the identifiers in the interface UNIT list was not declared in the body of the interface.

374 Program Identifier Expected

An identifier is missing after the keyword PROGRAM or MODULE. This is a panic error!

375 UNIT Identifier Expected

The unit identifier is missing after the phrase "IMPLEMENTATION OF". This is a panic error!

376 Program Not Found

The compiler expects, but cannot find, one of the reserved words PROGRAM, MODULE, or IMPLEMENTATION OF. This is a panic error! (This error can occur if the source file is not a compiland.)

377 File End Expected Skip To End

The compiler found addition source text after what appeared to be the end and ignored everything after what it thought was the end.

378 Program Not Found

The compiler expects, but cannot find, the main body of a compiland or the final END.

COMPILER BACK END ERRORS

The main source of back end errors is user error from either the optimizer or the code generator. There are, in fact, very few of these errors. All are concerned with limitations that cannot be detected by the front end.

Back end errors cause an immediate abort, while an error number and approximate listing line number appear on your screen.

The back end errors are listed below:

Code	Message
1	Attempt to divide by zero. For example, A DIV Ø.
2	Overflow during integer constant folding. For example, MAXINT + A + MAXINT.
3	Expression too complex/Too many internal labels. Try breaking up expression with intermediate value assigns.
4	Too many procedures and/or functions. Try breaking up compiland into modules or units.
5	Range error (number too large to fit into target).

COMPILER INTERNAL ERRORS

All errors labeled "Compiler" in Section "Compiler Front End Errors," are compiler internal errors that should never occur. In the event that one does occur, please report it to Burroughs Corporation immediately.

The back end of the compiler also makes a large number of internal consistency checks. These checks should always be correct and never give an internal error.

When they do occur, back end internal error messages have the following format:

```
*** Internal Error NNN
```

NNN is the internal error number, which ranges from 1 to 999. There is little you can do when an internal error occurs, except report it and perhaps modify your program near the line where the error occurred.

RUNTIME ERROR MESSAGES

Errors detected at runtime are either file system errors or other program exceptions. File system errors are described first.

FILE SYSTEM ERRORS (1000-1099)

File system error codes range from 1ØØØ to 1Ø99. Error codes go into the ERRC field of the file control block. File system errors are reported in the following format:

```
? Error: <error type> error in file <filename>
  Error Code <error code>, System status <status code>
  PC = <program counter>, FP = <frame pointer>, SP = <stack
       pointer>
```

If <error code> is in the range 1ØØØ - 1Ø99, then the error was detected by the BTOS Operating System and <status code> is a BTOS status code. See Appendix A of the B 2Ø Operating System (BTOS) Reference Manual.

File system errors all have the format,

```
<error type> error in file <filename>
```

followed by the error code. The <error type> field is based on the ERRS field of the file control block, as follows:

Code	Message
Ø	No error
1	Hard Data Hard data error (parity, CRC, check sum, etc.).
2	Device Name Invalid unit/device/volume name format or number.

3	Operation	Invalid operation: GET if EOF, RESET a printer, etc.
4	File System	File system internal error, ERRS > 15, etc.
5	Device Offline	Unit/device/volume no longer available.
6	Lost File	File itself no longer available.
7	File Name	Invalid syntax, name too long, no temporary names, etc.
8	Device Full	Disk or directory full.
9	Unknown Device	Unit/device/volume not found.
1Ø	File Not Found	File itself not found.
11	Protected File	Duplicate filename; write-protected.
12	File In Use	File in use, concurrency lock, already open.

13 File Not Open

File closed, I/O to unopen FCB.

14 Data Format

Data format error, decode error, range error.

15 Line Too Long

Buffer overflow, line too long.

Runtime File System (1100-1199)

If <error code> is in the range of 1100 - 1999, then the error was detected by the Pascal file system. These errors are explained below.

File System Errors (1100-1199)

Code	Message
1100	ASSIGN or READFN of filename to open file This error is only caught for textfiles.
1101	Reference to buffer variable of closed textfile
1102	Textfile READ or WRITE call to closed file
1103	READ when EOF is true (SEQUENTIAL mode)
1104	READ to REWRITE file, or WRITE to RESET file (SEQUENTIAL mode)
1105	EOF call to closed file
1106	GET call to closed file
1107	GET call when EOF is true (SEQUENTIAL mode)
1108	GET call to REWRITE file (SEQUENTIAL mode)
1109	PUT call to closed file
1110	PUT call to RESET file (SEQUENTIAL mode)

1111	Line too long in DIRECT textfile
1112	Decode error in textfile READ BOOLEAN
1113	Value out of range in textfile READ CHAR
1114	Decode error in textfile READ INTEGER
1115	Decode error in textfile READ SINT (integer subrange)
1116	Decode error in textfile READ REAL
1117	LSTRING target not big enough in READSET
1118	Decode error in textfile READ WORD
1119	Decode error in textfile READ BYTE (word subrange)
1120	SEEK call to closed file
1121	SEEK call to file not in DIRECT mode
1122	Encode error (field width > 255) in textfile WRITE BOOLEAN
1123	Encode error (field width > 255) in textfile WRITE INTEGER
1124	Encode error (field width > 255) in textfile WRITE REAL
1125	Encode error (field width > 255) in textfile WRITE WORD
1126	Decode error (field width > 255)in textfile READ INTEGER4
1127	Encode error (field width > 255)in textfile WRITE INTEGER4

OTHER RUNTIME ERRORS (2000-2999)

Nonfile system error codes range from 2000 to 2999. In some cases, metacommands control whether or not the compiler checks for the error. In other cases, the compiler always checks. The list below indicates which, if any, metacommand controls the error checking.

Memory Errors (2000-2999)

Since the stack and the heap grow toward each other, all memory errors are related; for example, a stack overflow can cause a "Heap Is Invalid" error if $STACKCK is off and the stack overflows.

Code	Message
2000	Stack Overflow
	The stack (frame) ran out of memory while calling a procedure or function. This condition is checked if the $STACKCK metacommand is on, and may be checked in some other cases.
2001	No Room In Heap
	The heap ran out of room for a new variable during the NEW (GETHQQ) procedure. This error is always caught.
2002	Heap Is Invalid
	During the NEW (GETHQQ) procedure, the allocation algorithm discovered the heap structure is wrong. This error is always caught.
2003	Heap Allocator Interrupted
	An interrupt procedure interrupted NEW (GETHQQ) and did a NEW call itself. The heap allocator modifies the heap, so it is a critical section. This error is not caught in all versions.
2004	Allocation Internal Error
	There was an unexpected error return when GETHQQ was requesting additional heap space from the operating system. Please report occurrences of this error to Burroughs Corporation.
2031	NIL Pointer Reference

DISPOSE or $NILCK+ found a pointer with a NIL (i.e., Ø) value.

2Ø32 Uninitialized Pointer

DISPOSE or $NILCK+ found an uninitialized (value 1) pointer. This only occurs if the metacommand $INITCK is on.

2Ø33 Invalid Pointer Range

DISPOSE or $NILCK+ found a pointer that does not point into the heap or is otherwise invalid. (It may have pointed to a disposed block that was removed from the heap and given back to the system.)

2Ø34 Pointer To Disposed Var

DISPOSE or $NILCK+ found a pointer to a heap block that has been disposed. Calling DISPOSE twice for the same variable is invalid.

2Ø35 Long DISPOSE Sizes Unequal

In a long form of DISPOSE, the actual length of the variable did not equal the length based on the tag values given.

Ordinal Arithmetic Errors (2050-2099)

Code **Message**

2Ø5Ø No CASE Value Matches Selector

In a CASE statement without an OTHERWISE clause, none of the branch statements had a CASE constant value equal to the selector expression value. This error is only checked if the $RANGECK metacommand is on.

2Ø51 Unsigned Divide By Zero

A WORD value was divided by zero. This error is checked only if the $MATHCK metacommand is on.

2052 Signed Divide By Zero

An INTEGER value was divided by zero. This error is checked only if the $MATHCK metacommand is on.

2053 Unsigned Math Overflow

A WORD result is outside the range zero to MAXWORD. This error is checked only if the $MATHCK metacommand is on.

2054 Signed Math Overflow

An INTEGER result is outside the range from -MAXINT to +MAXINT. This error is checked only if the $MATHCK metacommand is on.

2055 Unsigned Value Out Of Range

The source value for assignment or value parameter is out of range for the target value. The target may be a subrange of WORD (including BYTE), or CHAR, or an enumerated type. This error can also occur in SUCC and PRED functions and when the length of an LSTRING is assigned. All of these conditions are checked if the $RANGECK metacommand is on.

The error also occurs when an array index is out of bounds and the array has an unsigned index type. This condition is checked when the $INDEXCK metacommand is on.

2056 Signed Value Out Of Range

This error is similar to message 2055, but applies to the INTEGER type and its subranges.

2057 Uninitialized 16 Bit Integer Used

Either an INTEGER or 16-bit INTEGER subrange variable is used without being assigned first, or such a variable has the invalid value of -32768. This condition is checked if the $INITCK metacommand is on.

2058 Uninitialized 8 Bit Integer Used

Either a SINT or 8-bit INTEGER subrange variable is used without being assigned first, or such a variable has the invalid value of -128. This condition is checked if the $INITCK metacommand is on.

Type REAL Arithmetic Errors (2100-2149)

Code	Message
2100	REAL Divide By Zero

A REAL value is divided by zero. This error is always caught.

2101	REAL Math Overflow

A REAL value is too large for representation. This error is always caught.

2104	SQRT of Negative Argument

The parameter for a square root function is less than zero. This error is always caught.

2105	LN of Non-Positive Argument

The parameter of a natural log function is less than or equal to zero. This error is always caught.

2106	TRUNC/ROUND Argument Range

The REAL parameter of a TRUNC, TRUNC4, ROUND, or ROUND4 function is outside the range of INTEGERs. This error is always caught.

2131	Tangent Argument Too Small

The parameter for a TANRQQ function is so small that the result is invalid. This error is always caught.

2132	Arcsin or Arccos of REAL > 1.0

The parameter of an ASNRQQ or ACSRQQ function is greater than one. This error is always caught.

2133 Negative Real To Real Power

The first argument of an PRDRQQ or PRSRQQ function is less than zero. This error is always caught.

2134 Real Zero To Negative Power

There was an attempt to raise zero to a negative power in one of the functions PISRQQ, PIDRQQ, PRDRQQ, or PRSRQQ.

2135 REAL Math Underflow

The significance of a REAL expression has been reduced to zero.

2136 REAL Indefinite (Uninitialized Or Previous Error)

The REAL value called "infinity" was encountered. This may occur if the $INITCK metacommand is on and an uninitialized REAL value is used, or if a previous error set a variable to indefinite as part of its masked error response.

2138 REAL IEEE Denormal Detected

A very small real number was generated and may no longer be valid due to loss of significance.

2139 Reserved

214Ø REAL Arithmetic Processor Instruction Illegal Or Not Emulated

An attempt was made to execute an illegal arithmetic coprocessor instruction, or the floating point emulator cannot emulate a legal coprocessor instruction.

Structured Type Errors (2150-2199)

Code	Message
215Ø	String Too Long in COPYSTR

The source string for a COPYSTR intrinsic function is too large for the target string. This error is always caught.

2151	Lstring Too Long In Intrinsic Procedure

The target LSTRING is too small in an INSERT, DELETE, CONCAT, or COPYLST intrinsic procedure. This error is always caught.

218Ø	Set Element Greater Than 255

The value in a constructed set exceeds the maximum of 255. This error is always caught.

2181	Set Element Out Of Range

The value in a set assignment or set value parameter is too large for the target set. This error is caught only if the $RANGECK metacommand is on.

INTEGER4 Errors (2200-2249)

Code	Message
22ØØ	Long Integer Divide By Zero

An INTEGER4 value is divided by zero. This error is always caught.

22Ø1	Long Integer Math Overflow

An INTEGER4 value is too large for representation. This error is always caught.

2234	Reserved

Other Errors (2400-2999)

Code	Message
2400	Reserved
2450	Unit Version Number Mismatch During unit initialization, the user (one with the USES clause) and implementation of an interface were discovered to have been compiled with unequal interface version numbers. This error is always caught.

APPENDIX E
SUMMARY OF RESERVED WORDS

Reserved words at the standard level:

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

Additional reserved words at the extended level:

```
BREAK               OTHERWISE
CONSTS              RETURN
CYCLE               UNIT
IMPLEMENTATION      USES
INTERFACE           VALUE
MODULE              VARS
                    XOR
```

Additional reserved words at the system level:

```
ADR
ADS
```

Names of attributes

```
EXTERN              PURE
EXTERNAL            READONLY
ORIGIN              STATIC
PUBLIC
```

Names of directives:

```
EXTERN
EXTERNAL
FORWARD
```

Logically, directives are reserved words. Since additional directives are allowed in ISO Pascal, all are included at the standard level. Note that EXTERN is both a directive and an attribute; EXTERNAL is a synonym for EXTERN in both cases.

APPENDIX F
SUMMARY OF AVAILABLE PROCEDURES AND FUNCTIONS

This appendix provides a summary listing of all available functions and procedures, along with the name of the group in which they are presented in Chapter 11, Available Procedures and Functions."

Name	Description	Category
ABORT	Terminate program	Extended level
ABS	Absolute value function	Arithmetic
ACDRQQ	REAL8 arc cosine function	Arithmetic
ACSRQQ	REAL4 arc cosine function	Arithmetic
AIDRQQ	REAL8 truncate function	Arithmetic
AISRQQ	REAL4 truncate function	Arithmetic
ANDRQQ	REAL8 round toward zero	Arithmetic
ANSRQQ	REAL4 round toward zero	Arithmetic
ARCTAN	Arc tangent function	Arithmetic
ASDRQQ	REAL8 arc sine function	Arithmetic
ASSRQQ	REAL4 arc sine function	Arithmetic
ASSIGN	Assign filename	File system
ATDRQQ	REAL8 arc tangent function	Arithmetic
ATSRQQ	REAL4 arc tangent (A/B)	Arithmetic
A2DRQQ	REAL8 arc tangent (A/B)	Arithmetic
A2SRQQ	REAL4 arc tangent function	Arithmetic
BEGOQQ	Initialize user	Library
BEGXQQ	Overall initialization	Library
BYLONG	WORD or INTEGER to INTEGER4	Extended level
BYWORD	Put bytes in word	Extended level
CHDRQQ	REAL8 hyperbolic cosine	Arithmetic
CHR	Get ASCII char of value	Data conversion
CHSRQQ	REAL4 hyperbolic cosine	Arithmetic
CLOSE	Close file	File system
CNDRQQ	REAL4 cosine function	Arithmetic
CNSRQQ	REAL4 cosine function	Arithmetic
CONCAT	Concatenate LSTRING	String
COPYLST	Copy to LSTRING	String
COPYSTR	Copy to STRING	String
COS	Cosine function	Arithmetic
DECODE	Decode LSTRING to variable	Extended level
DELETE	Remove portion of LSTRING	String
DISCARD	Close and delete file	File system
DISPOSE	Dispose of heap item	Dynamic allocation
ENCODE	Encode expression to LSTRING	Extended level
ENDOQQ	User termination	Library
ENDXQQ	Program termination	Library
EOF	Boolean end-of-file	File system
EOLN	Boolean end-of-line	File system
EVAL	Evaluate functions	Extended level
EXDRQQ	REAL8 exponential function	Arithmetic
EXP	Exponential function	Arithmetic
EXSRQQ	REAL4 exponential function	Arithmetic
FILLC	Fill area with C, relative	System level
FILLSC	Fill area with C, segmented	System level
FLOAT	Convert INTEGER to REAL	Data conversion

FLOAT4	Convert INTEGER4 to REAL	Data conversion
GET	Get next file component	File system
HIBYTE	Get high BYTE	Extended level
HIWORD	Get high WORD	Extended level
INSERT	Insert string	String
LADDOK	32-bit signed addition check	Library
LDDRQQ	REAL8 log base ten function	Arithmetic
LDSRQQ	REAL4 log base ten function	Arithmetic
LMULOK	32-bit signed multiply check	Arithmetic
LN	Natural log function	Arithmetic
LNDRQQ	REAL8 natural log	Arithmetic
LNSRQQ	REAL4 natural log	Arithmetic
LOBYTE	Get low BYTE	Extended level
LOWER	Get lower bound	Extended level
LOWORD	Get low WORD	Extended level
MDDRQQ	REAL8 modulo function	Arithmetic
MDSRQQ	REAL4 modulo function	Arithmetic
MNDRQQ	REAL8 minimum function	Arithmetic
MNSRQQ	REAL4 minimum function	Arithmetic
MOVEL	Move bytes left, relative	System level
MOVER	Move bytes right, relative	System level
MOVESL	Move bytes left, segmented	System level
MOVESR	Move bytes right, segmented	System level
MXDRQQ	REAL8 maximum function	Arithmetic
MXSRQQ	REAL4 maximum function	Arithmetic
NEW	Allocate new heap item	Dynamic allocation
ODD	Boolean odd function	Data conversion
ORD	Get ordinal value	Data conversion
PACK	Pack CHAR array	Data conversion
PAGE	Write new page	File System
PIDRQQ	REAL8 to INTEGER power	Arithmetic
PISRQQ	REAL4 to INTEGER power	Arithmetic
POSITN	Find position of substring	String
PRED	Predecessor function	Data conversion
PRDRQQ	REAL8 to REAL8 power	Arithmetic
PRSRQQ	REAL4 to REAL4 power	Arithmetic
PUT	Put value to file	File system
READ	Read file	File system
READFN	Read filename	File system
READLN	Read file to end of line	File system
READSET	Read set	File system
RESET	Ready file for read	File system
RESULT	Return result of function	Extended level
RETYPE	Force expression to type	System level
REWRITE	Ready file for write	File system
ROUND	Round REAL	Data conversion
ROUND4	Round INTEGER4	Data conversion
SADDOK	16-bit signed addition check	Library
SCANEQ	Scan until char found	String
SCANNE	Scan until char not found	String
SEEK	Position at direct file record	File system
SHDRQQ	REAL8 hyperbolic sine	Arithmetic
SHSRQQ	REAL4 hyperbolic sine	Arithmetic
SIN	Sine function	Arithmetic

SIZEOF	Get size of structure	Extended level
SMULOK	16-bit signed multiply check	Library
SNDRQQ	REAL8 sine function	Arithmetic
SNSRQQ	REAL4 sine function	Arithmetic
SQR	Square function	Arithmetic
SQRT	Square root function	Arithmetic
SRDRQQ	REAL8 square root	Arithmetic
SRSRQQ	REAL8 square root	Arithmetic
SUCC	Successor function	Data conversion
THDRQQ	REAL8 hyperbolic tangent	Arithmetic
THSRQQ	REAL4 hyperbolic tangent	Arithmetic
TNDRQQ	REAL8 tangent function	Arithmetic
TNSRQQ	REAL4 tangent function	Arithmetic
TRUNC	Truncate REAL	Data conversion
TRUNC4	Truncate INTEGER4	Data conversion
UADDOK	Unsigned addition check	Library
UMULOK	Unsigned multiply check	Library
UNPACK	Unpack STRING to array	Data conversion
UPPER	Get upper bound	Extended level
WRD	Convert to WORD value	Data conversion
WRITE	Write file	File system
WRITELN	Write line to file	File system

APPENDIX G
SUMMARY OF METACOMMANDS

This appendix provides a single alphabetical list of all of the metacommands described in Chapter 4, "Metacommands." Defaults, if any, are shown following the metacommand.

Metacommand	Action
$BRAVE+	Sends messages to the terminal screen.
$DEBUG-	Turns on or off all error checking (CK).
$ENTRY-	Generates procedure entry and exit calls for debugger.
$ERRORS:25	Sets number of errors allowed per page.
$GOTO-	Flags GOTOs as "considered harmful."
$IF <constant> $THEN <text1> $ELSE <text2> $END	Allows conditional compilation of <text1> source if <constant> is greater than zero.
$INCLUDE:'<file>'	Switches compilation to file named.
$INCONST	Allows interactive setting of constant values at compile time.
$INDEXCK+	Checks for array index values in range.
$INITCK-	Checks for use of uninitialized values.
$INTEGER	Sets the length of the INTEGER type.
$LINE-	Generates line number calls for debugger.
$LINESIZE:79	Sets width of source listing.
$LIST+	Turns on or off source listing.
$MATHCK+	Checks for mathematical errors.
$MESSAGE	Displays a message on terminal screen.
$NILCK+	Checks for bad pointer values.
$OCODE+	Turns on or off object code listing.
$PAGE+	Skips to next page.
$PAGE:<n>	Sets page number for next page.

$PAGEIF:<n>	Skips to next page if less than <n> lines left.
$PAGESIZE:55	Sets page length of source listing.
$POP	Restores saved value of all metacommands.
$PUSH	Saves current value of all metacommands.
$RANGECK+	Checks for subrange validity.
$REAL	Sets the length of the REAL type.
$ROM	Warns on static initialization.
$RUNTIME-	Determines context of runtime errors.
$SIMPLE	Disables global optimizations.
$SIZE	Minimizes size of code generated.
$SKIP:<n>	Skips <n> lines or to end of page.
$SPEED	Minimizes execution time of code.
$STACKCK+	Checks for stack overflow at entry.
$SUBTITLE:'<subt>'	Sets page subtitle.
$SYMTAB+	Sends symbol table to source listing.
$TITLE:'<title>'	Gives page title for source listing.
$WARN+	Gives warning messages in source listing.

APPENDIX H
EXTENDED PASCAL FEATURES AND THE ISO STANDARD

CONTENTS

ENTENDED PASCAL AND THE ISO STANDARD

The ISO standard defines a large number of error conditions, but allows a particular implementation to handle an error by documenting the fact that the error is not caught. These "errors not caught," and other differences between this extended Pascal and the ISO standard, are described below.

Extended Pascal allows the following minor extensions to the current ISO/ANSI/IEEE standard:

1. a question mark (?) and a at-sign (@) are substitutes for the up arrow (^)

2. the underscore (_) in identifiers

Due to the way the compiler binds identifiers, the new reserved words added at the extended and system levels cannot be used as identifiers at the standard level. A new directive, EXTERN, and new predeclared functions are standard in this extended Pascal.

The differences between the standard level of this Pascal and the current ISO/ANSI/IEEE standard are summarized as follows:

1. The ISO standard requires a separator between numbers and identifiers or keywords.

   In some cases, this extended Pascal doesn't require a separator between a number and an identifier or keyword, e.g., "1ØØmod" is accepted as "1ØØ mod" without error.

2. The ISO standard does not allow passing a component of a PACKED structure as a reference parameter.

   This extended Pascal specifically permits passing a CHAR element of a PACKED ARRAY [1..n] OF CHAR as a reference parameter. Passing a tag field as a reference is an error not caught. Passing other packed components gives the usual error.

3. The ISO standard does not include the textfile line-marker character in the set of CHAR values.

   This extended Pascal permits all 256 8-bit values as CHAR values.

4. The ISO standard requires a variant to be given for all possible tag values.

   This extended Pascal permits a variant record

declaration in which not all tag values are given.

5. The ISO standard requires that an identifier have only one meaning in any scope.

   In this extended Pascal, using an identifier and then redeclaring it in the same scope is an error not caught. For example, the following,

```
CONST X=Y;  VAR Y:  CHAR;
```

   has two meanings for Y in the same scope. This Pascal generally uses the latest definition for an identifier. There is one ambiguous case: If you declare type FOO in one scope and in an inner scope TYPE P = ^ FOO; FØØ = type; then FOO has two meanings and intent is ambiguous. In this case, the compiler uses the later definition of FOO and issues a warning.

6. The ISO standard requires field width "M" to be greater than zero in WRITE and WRITELN procedures.

   This extended Pascal treats M < Ø as if M = ABS(M), but field expansion takes place from the right rather than the left. M can also be zero, to WRITE nothing. Textfile WRITE(LN) parameters can take both M and N parameters (ignored if not needed). The form "V::N" is allowed. When writing an INTEGER, the N parameter sets the output radix; when writing an enumerated type, the N parameter sets the ordinal number or constant identifier option.

7. The ISO standard does not allow a variable created with the long form of NEW to be assigned, used in an expression, or passed as a parameter. However, this is difficult to check for at compile time and expensive to check at runtime.

   This extended Pascal allows assignments to these variables using the actual length of the target variable. The ISO standard error is not caught.

8. The ISO standard does not allow the short form of DISPOSE to be used on a structure allocated with the long form of NEW. The ISO standard only permits a variable allocated with the long form of NEW to be released with the long form of DISPOSE, and all tag fields should never change between the calls.

   This extended Pascal allows the short form of DISPOSE

to be used on a structure allocated with the long form of NEW, and does not check for changes in tag values.

9. The ISO standard declares that when a "change of variant" occurs (such as when a new tag value is assigned), all the variant fields become undefined.

   This extended Pascal does not set the fields uninitialized when a new tag is assigned and so does not catch use of a variant field with an undefined value.

10. The ISO standard does not allow a variable with an active reference (i.e., the records of an executing WITH statement or an actual reference parameter) to be disposed (if a heap variable) or changed by a GET or PUT (if a file buffer variable).

    This Pascal does not catch these as errors.

11. The ISO standard currently defines I MOD J as an error if J < 0 and the result of MOD is positive, even if I is negative.

    This extended Pascal does not currently use the new draft standard semantics for the MOD operator. Programs intended to be portable should not use MOD unless both operands are positive.

12. The ISO standard at Level 1 defines conformant array.

    This extended Pascal does not implement the conformant array concept in Level 1 of the ISO standard. Super arrays provide much the same functionality in a more flexible way.

13. The ISO standard requires the control variable of a FOR loop to be local to the immediate block. Any assignment to this control variable is an error.

    This extended Pascal allows nonlocal variable to be used if it is STATIC, so either a local variable or one at the PROGRAM level can be a FOR statement control variable. This Pascal also does not detect an assignment to the control variable as an error if assignment occurs in a procedure or function called within the FOR statement.

14. The ISO standard requires the CHR argument to be INTEGER.

    This extended Pascal allows CHR to take any ordinal type.

SUMMARY OF EXTENDED PASCAL FEATURES

The following summarizes this Pascal's extensions to the ISO standard. Unless otherwise noted, all are at the extended level.

Syntactic and Pragmatic Features

1. the metalanguage (standard level)

```
$BRAVE                    $PAGE
$DEBUG                    $PAGEIF
$ENTRY                    $PAGESIZE
$ERRORS                   $POP
$GOTO                     $PUSH
$INCLUDE                  $RANGECK
$INCONST                  $REAL
$INDEXCK                  $ROM
$INTICK                   $RUNTIME
$IF $THEN $ELSE $END      $SIZE
$INTEGER                  $SKIP
$LINE                     $SPEED
$LINESIZE                 $STACKCK
$LIST                     $SUBTITLE
$MATHCK                   $SYMTAB
$MESSAGE                  $TITLE
$NILCK                    $WARN
$OCODE
$OPTBUG
```

2. extra listing (standard level)

   a. flags for jumps, globals, identifier level, control level, header, trailer

   b. textual error and warning messages

3. syntactic additions

   a. ! as comment to end of line

   b. square brackets equivalent to BEGIN/END

4. nondecimal number notation

   a. numeric constants with # or nn# (where nn = 2..36)

   b. DECODE/READ takes # notation

   c. ENCODE/WRITE with N of 2, 8, 1Ø, 16

5. extended CASE range

   a. for CASE statements and record variants

   b. OTHERWISE for all other values except records

   c. A..B for range of values

Data Types and Modes

1. WORD type, WRD function, MAXWORD constant
2. REAL4 and REAL8 types
3. INTEGER4 type, MAXINT4 const;
4. FLOAT4, ROUND4, and TRUNC4 functions
5. Address types (system level)

   a. ADR and ADS types and operators

   b. VARS and CONSTS parameters

6. SUPER array types

   a. conformant parameters

   b. dynamic length heap variables

   c. multidimensional super arrays

   d. STRING and LSTRING super types

7. LSTRING type, NULL constant, .LEN field
8. Explicit byte offsets in records (system level)
9. CONST and CONSTS reference parameters for constants and expressions

10. Structured (array, record, and set) constants

11. Extended functions returning any assignable type

12. Variable selection on values returned from functions

13. Attributes

EXTERN	PURE
EXTERNAL	READONLY
ORIGIN	STATIC
PUBLIC	

Operators and Intrinsics

1. extended level operators:

   a. bitwise logical: AND OR NOT XOR

   b. set operators: < >

2. constant expressions:

   a. string constant concatenation with * operator

   b. numeric, ordinal, Boolean expressions in type clauses

   c. other constant functions:

CHR	UPPER
DIV	WRD
HIBYTE	*
HIWORD	+
LOBYTE	-
LOWER	<
LOWORD	<=
MOD	<>
ORD	=
RETYPE	>
SIZEOF	>=

3. additional intrinsic functions at extended level:

ABORT	LOWORD
BYLONG	RESULT
BYWORD	SIZEOF
DECODE	UPPER

ENCODE	HIWORD
EVAL	LOBYTE
HIBYTE	LOWER

4. additional intrinsic functions at system level:

FILLC	MOVESL
FILLSC	MOVESR
MOVEL	RETYPE
MOVER	

5. intrinsic functions that operate on strings:

   a. for STRING or LSTRING: COPYSTR POSITN SCANEQ SCANNE

   b. for LSTRING only: CONCAT INSERT DELETE COPYLST

6. REAL library functions (standard level)

7. Pascal library functions (standard level):

BEGOQQ	LMULOK
BEGXQQ	SADDOK
ENDOQQ	SMULOK
ENDXQQ	UADDOK
LADDOK	UMULOK

Control Flow and Structure Features

1. control flow statements: BREAK, CYCLE, and RETURN
2. sequential control operators: AND THEN and OR ELSE in IF, WHILE, REPEAT
3. extended FOR loop: FOR VAR variable
4. VALUE section to initialize static variables
5. mixed order LABEL, CONST, TYPE, VAR, VALUE sections
6. compilable MODULES, with global attributes
7. UNIT INTERFACE and IMPLEMENTATION:
   a. interface version numbers, version checking
   b. optional rename of constituents

c. guaranteed unique unit initialization

d. optional unit initialization

Extended Level I/O and Files

1. textfile line length declaration, TEXT (nnn)
2. READ enumerated, Boolean, pointer, STRING, LSTRING
3. WRITE enumerated, pointer, LSTRING
4. negative M value to justify left instead of right
5. temporary files
6. DIRECT mode files, SEEK procedure
7. ASSIGN, CLOSE, DISCARD, READSET, READFN procedures
8. FILEMODES type and constants, F.MODE access
9. error trapping, F.TRAP and F.ERRS access

System Level I/O

This Pascal's extensions to the ISO standard offers full FCBFQQ type equivalent to FILE types

APPENDIX I
CONTROL OF THE VIDEO DISPLAY

CONTENTS

A Pascal program can control the video display by writing a multibyte escape sequence to the video display. In this way, a program can:

- control character attributes (blinking, reverse video, underscoring, half bright),
- control screen attributes (reverse video, half bright),
- control cursor positioning and visibility,
- fill a rectangle with a single character,
- control scrolling of lines,
- direct video display output to any frame,
- control pausing between full frames of data,
- control the keyboard LED indicators, and
- erase to the end of the current line or frame.

A multibyte escape sequence consists of the video display escape character, a command character, and parameters. The video display escape character is CHR(255). To print an escape character, precede it with another escape character.

ERROR CONDITIONS IN ESCAPE SEQUENCES

An escape character sequence is in error if the command characters or parameters are unrecognized or the parameters are inconsistent.

The following program turns on the cursor, writes the message "This is a test", and waits for input:

```
PROGRAM Test (INPUT, OUTPUT);
VAR
   ls : LSTRING (128);

BEGIN
   ls := CHR(255) * 'vn';
   Write (ls, 'This is a test');
   Readln;
END.
```

Video Display Coordinates

Pascal interprets some parameters as x and y coordinates on the video display.

A value of 255 for x or y specifies, respectively, the last column or line of the frame.

If the value of x or y is less than 255 and greater than the last column or line, then the escape sequence is in error.

The concatenation operator "*" can only be used to create constant string expressions. Therefore the following code would be incorrect:

```
VAR  i  : INTEGER; str : STRING(4);
str := CHR(255) * 'C' * CHR(0) * CHR(i)
```

The above code is incorrect because CHR(i) is not constant, but rather varies with i. To create variable string expressions with concatenation, you may use the LSTRING intrinsic "CONCAT".

CONTROLLING CHARACTER ATTRIBUTES: THE 'A' COMMAND

Format 1: CHR (255) * 'A<parameter>'

where <parameter> is a character in the range A to P.

Format 2: CHR(255) * 'AZ'

Purpose: Format 1 is used to enable or disable character attributes for characters following the escape sequence. The table below shows the attributes enabled or disabled for each escape sequence using the 'A' command.

An x in the table below indicates that the attribute is enabled; otherwise, it is disabled. The character attributes are: blinking (B), reverse video (R), underlining (U), and half bright (H).

	B	R	U	H
CHR(255) * 'AA'				
CHR(255) * 'AB'				x
CHR(255) * 'AC'			x	
CHR(255) * 'AD'			x	x
CHR(255) * 'AE'		x		
CHR(255) * 'AF'		x		x
CHR(255) * 'AG'		x	x	
CHR(255) * 'AH'		x	x	x
CHR(255) * 'AI'	x			
CHR(255) * 'AJ'	x			x
CHR(255) * 'AK'	x		x	
CHR(255) * 'AL'	x		x	x
CHR(255) * 'AM'	x	x		
CHR(255) * 'AN'	x	x		x
CHR(255) * 'AO'	x	x	x	
CHR(255) * 'AP'	x	x	x	x

Format 2 is used to enable a mode whereby writing a character into a character position does not change the character attributes of that character position.

CONTROLLING SCREEN ATTRIBUTES: THE 'H' AND 'R' COMMANDS

Format 1: CHR(255) * 'H <parameter> '

where <parameter> is N or F.

Format 2: CHR(255) * 'R <parameter> '

where <parameter> is N or F.

Purpose: Format 1 is used to turn the half bright attribute on if the <parameter> is N. It is used to turn the half bright attribute off if the <parameter> is F.

Format 2 is used to turn the reverse video attribute on if the <parameter> is N. It is used to turn the reverse video attribute off if the <parameter> is F.

CONTROLLING CURSOR POSITION AND VISIBILITY: THE 'C' AND 'V' COMMANDS

Format 1:

```
CHR(255) * 'C'
* CHR(<Xposition>) * CHR(<Yposition>)
```

where <Xposition> and <Yposition> are integer expressions.

Format 2:

```
CHR(255) *'V<parameter>'
```

where <parameter> is N or F.

Purpose

Format 1 is used to position the cursor at coordinates (<Xposition>, <Yposition>).

Format 2 is used to make the cursor visible if the <parameter> is N. It is used to make the cursor invisible if the <parameter> is F.

FILLING A RECTANGLE: THE 'F' COMMAND

Format:

```
CHR(255) * 'F' *  character
* CHR(<Xposition>) * CHR(<Yposition>)
* CHR(<width>) * CHR(<height>)
```

where <character> is any character; <Xposition>, <Yposition>, <width>, and <length> are integer expressions.

Purpose:

The 'F' command is used to fill a rectangle on the video display with <character>. The currently enabled character attributes are given to each character in the rectangle. A <character> always specifies a character in the standard character set.

The coordinates (<Xposition>, <Yposition>) specify the upper left corner of the rectangle. A value of 255 for <width> and <height> specifies, respectively, the remaining width or height of the frame.

CONTROLLING LINE SCROLLING: THE 'S' COMMAND

Format:

```
CHR(255) * 'S'
* CHR(<firstline>) * CHR(<lastline>)
* CHR(<count>) * '<direction>'
```

where <direction> is D or U.

Purpose:

If the <direction> is D, the 'S' command is used to scroll down a portion of the frame beginning at line firstline and extending to (but not including) <lastline>. The <count> lines are scrolled and the top <count> lines of the frame portion are filled with blanks.

If the <direction> is U, the 'S' command is used to scroll up a portion of the frame beginning at line <lastline> and extending to (but not including) <firstline>. The <count> lines are scrolled and the bottom <count> lines of the frame portion are filled with blanks.

DIRECTING VIDEO DISPLAY OUTPUT: THE 'X' COMMAND

Format:

```
CHR(255) * 'X' * CHR(<frame>)
```

Purpose:

The 'X' command is used to direct video output to the <frame> 'th frame of the video display.

If the <frame> is 1, the 'X' command is used to direct video output to the Status Frame at the top of the video display.

CONTROLLING PAUSING BETWEEN FULL FRAMES: THE 'P' COMMAND

Format:

```
CHR(255) * 'P<parameter>'
```

where <parameter> is N or F.

Purpose:

If the <parameter> is N, the 'P' command is used to enable the pause facility. When the pause facility is enabled and further output to the frame would cause data to be scrolled off the top of the frame, the message:

Press NEXT PAGE to continue

is displayed on the last line of the frame.

If the <parameter> is F, the 'P' command is used to disable the pause facility.

CONTROLLING THE KEYBOARD LED INDICATORS: THE 'I' COMMAND

Format:

```
CHR(255) * 'I<parameter> N'   or
CHR(255) * 'I<parameter>F'
```

where <parameter> is 1, 2, 3, 8, 9, 0, or T.
'I<parameter>N' turns ON the led.
'I<parameter>F' turns OFF the led.

Purpose:

The 'I' command is used to turn on an LED indicator on the keyboard according to the table below.

parameter	Key
1	F1
2	F2
3	F3
8	F8
9	F9
0	F10
T	OVERTYPE

ERASING TO THE END OF THE LINE OR FRAME: THE 'E' COMMAND

Format:

```
CHR(255) * 'E<parameter>'
```

where <parameter> is L or F.

Purpose:

If the <parameter> is L, the 'E' command is used to erase to the end of the line.

If the <parameter> is F, the 'E' command is used to erase to the end of the frame.

Erases sets characters to spaces and turns off all character attributes.

FURTHER DETAILS

For a more detailed, language-independent explanation of video escape sequences, consult "Video Byte Streams" in the "Sequential Access Method" section of the B 20 Operating System Manual.

Documentation Evaluation Form

Title: B 20 Systems Pascal Reference Manual

Form No: 1162955

Date: August, 1983

Burroughs Corporation is interested in receiving your comments and suggestions regarding this manual. Comments will be utilized in ensuing revisions to improve this manual.

Please check type of Suggestion:

☐ Addition ☐ Deletion ☐ Revision ☐ Error

Comments:

From:

Name _____

Title _____

Company _____

Address _____

_____

Phone Number _____ Date _____

Remove form and mail to:

Documentation Dept, - East
Burroughs Corporation
Box CB7
Malvern, PA 19355