MS-PASCAL
VOLUME II

VICTOR Publications
380 El Pueblo Road
Scotts Valley, CA 95066
(408) 438-6680

NOTICE

VICTOR makes no representations or warranties of any kind whatsoever with respect to the contents hereof and specifically disclaims any implied warranties of merchantability or fitness for any particular purpose. VICTOR shall not be liable for errors contained herein or for incidental or consequential damages in connection with the furnishing, performance, or use of this publication or its contents.

VICTOR reserves the right to revise this publication from time to time and to make changes in the content hereof without obligation to notify any person of such revision or changes.

First VICTOR printing April, 1983.

ISBN: 0-88182-027-X Printed in U.S.A.

MS-PASCAL VOLUME I

User's Guide

Reference Manual

MS-PASCAL VOLUME II

Reference Manual (Continued)

11. EXPRESSIONS

Expressions are constructions that evaluate to values. Table 11-1 illustrates a variety of expressions that, if A = 1 and B = 2, evaluate to the value shown.

Table 11-1: Expressions

EXPRESSION	VALUE
2	2
A	1
A + 2	3
(A + 2)	3
(A + 2) * (B - 3)	-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.

The available operators are shown, in the order in which they are executed, in Table 11-2.

Table 11-2: Operators

TYPE	OPERATORS
Unary	NOT [ADR ADS]
Multiplying	* / DIV MOD AND (ISR SHL SHR)

Adding	+ - OR (XOR)
Relational	= <> <= >= < > IN

Operators shown in parentheses are available only at the extend level of MS-Pascal, those in brackets only at the system level.

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
```

Use parentheses to change operator precedence. The following expression evaluates to 9 rather than 7:

```
(1 + 2) * 3
```

If the $SIMPLE switch 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. The semantics of the precedence relationships are retained, but normal associative and distributive laws are used. For example, X * 3 + 12 is an optimization of 3 * (6 + (X - 2)).

Optimizations may occasionally give you unexpected overflow errors. For example,

```
(I - 100) + (J - 100)
```

is optimized into the following:

```
(I + J) - 200
```

An overflow error may result, although the original

expression did not yield an error (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. Normally, expressions are evaluated, except as noted in the discussion in the following pages.

A Pascal 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 MS-Pascal operators only apply to the following types:

INTEGER	INTEGER4	REAL
WORD	BOOLEAN	SET

The relational operators listed in Table 11-2 also apply for the CHAR, enumerated, string, and reference types. For all operators (except the set operator IN), operands must have compatible types.

11.1 SIMPLE TYPE EXPRESSIONS

As a rule, the operands and the value resulting from an operation are all 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. Be careful when mixing INTEGER and WORD constants in expressions. For example, if

CBASE is the constant 16#C000 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#C000 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:

- From INTEGER to REAL or INTEGER4
- 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 permitted. 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) is illegal.

(-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        {Illegal with REAL operands.}
123 DIV  5  =  24
```

1.3 DIV 5 {Illegal with REAL operands.}

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

MS-Pascal differs from the current draft ISO standard with respect to the semantics for DIV and MOD with negative operands, 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 extend level 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 1's complement operation on the single operand. If an INTEGER variable V has the value MAXINT, NOT V gives the illegal INTEGER value -32768. This generates an error if the initialization switch 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
```

5. SHL SHR ISR

These extend level operators provide bitwise shifting functions.

SHL and SHR are logical shifts left and right. ISR is an integer (signed) arithmetic shift right: the sign bit is always propagated, even on a WORD type operand. Since the compiler cannot generate a simple right shift for INTEGER division (-1 DIV 2 would be incorrect) and division is a very time-consuming operation, SHR or ISR can be used instead of DIV where appropriate.

Operands must be both INTEGER, both WORD, or both INTEGER4; they cannot be REAL. The result has the same type as the operands.

The left operand is shifted, and the right operand is the shift count in bits. A shift count less than 0 or greater than 32 produces undefined results and generates an error message if the range checking switch is on.

Shifts never cause overflow errors; shifted bits are simply lost.

Given that X = 2#1111111100000000, the shifting functions perform the following operations:

```
X          1111111100000000

X SHL 1    1111111000000000

X SHR 1    0111111110000000

X ISR 1    1111111110000000 {sign extension}
```

11.2 BOOLEAN EXPRESSIONS

The Boolean operators at the standard level of MS-Pascal are:

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

XOR is available at the extend and systems levels.

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. The following example illustrates the danger of assuming that they don't:

```
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 run-time error. This evaluation may or may not take place. Sometimes both operands are evaluated during

optimization, and sometimes the evaluation of one causes the evaluation of the other to be skipped. In the latter case, either operand may be evaluated first.

Instead of the preceding example, use the following construction:

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

See Section 12.3.5 for information on using AND THEN and OR ELSE to handle situations, such as in the previous example, where tests are examined sequentially.

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

```
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 if 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 six relational operators (=, <>, <=, >=, <, and >) have their normal meaning when applied to numeric, enumerated, CHAR, or string operands. Section 11.3 discusses the meaning of these relational operators (along with the relational operator IN) when applied to sets. Since the relational operators in Boolean expressions have a lower precedence than AND and OR, the following is incorrect:

```
IF I < 10 AND J = K THEN
```

Instead, you must write:

```
IF (I < 10) AND (J = K) THEN
```

Also, you cannot use the numeric types where a Boolean operand is called for. (Some other languages permit this.) For an integer I, the clause IF I THEN is illegal; you must use the following instead:

```
IF I <> 0 THEN
```

Note, however, that MS-Pascal does allow the following:

```
$IF I $THEN
```

11.3 SET EXPRESSIONS

Table 11-3 shows the MS-Pascal operators that apply differently to sets than for other types of expressions.

Table 11-3: Set Operators

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 legally be outside the range of the base type of B. For example, X IN B is always false if the following statements are true:

```
X = 1
B = SET OF 2..9
```

(1 is compatible, but not assignment compatible, with 2..9).

Angle brackets are set operators only at the extend level of MS-Pascal, since the ISO standard 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 can 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 illegal 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 compatible only with sets that are PACKED or unpacked, respectively, in the same way as the explicit type of the constant.

11.4 FUNCTION DESIGNATORS

A function designator specifies the activation of a function. It consists of the function identifier, followed by a (possibly empty) list of "actual

parameters" in parentheses:

```
{Declaration of the function ADD.}
FUNCTION ADD (A, B: INTEGER); INTEGER;
  .
  .
{Use of the function ADD in an expression.}
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. (See Section 13.4 for more information on parameters.)

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.

In most computer languages, 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."

At the standard level, a function can return either a simple type or a pointer. At the extend level, a function can return any assignable type (any type

except a file or super array).

At the standard level, a pointer returned by a function can be compared, assigned, or passed only as a value parameter. At the extend level, however, the usual selection syntax for reference types, arrays, and records is allowed, following the function designator. See Section 10.4 for information.

Examples of function designators:

```
SIN (X+Y)

NEXTCHAR

NEXTREC (17) ^
{Here the function return type}
{is a pointer, and the returned}
{pointer value is dereferenced.}

NAD.NAME [1]
{Here the function has no parameters.}
{The return type is a record, one}
{field of which is an array.}
{The identifier for that field is}
{NAME. The example above selects}
{the first array component of the}
{returned record.}
```

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.

11.5 EVALUATING EXPRESSIONS

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), you must use FOO (I, (J+14)) 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 occur only 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, such as a constant or an expression in parentheses, 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.

Finally, 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, 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.

11.6 OTHER FEATURES OF EXPRESSIONS

EVAL and RESULT are two procedures available at the extend level 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 within a nested procedure or function.

At the system level, the function RETYPE allows you to change the type of a value.

11.6.1 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 useful only 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)
```

11.6.2 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;
```

11.6.3 THE RETYPE FUNCTION

Occasionally, you need to change the type of a value. You can do this with the RETYPE function, available at the system level of MS-Pascal. If the new type is a structure, RETYPE can be followed by the usual selection syntax. Use RETYPE with caution: 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}
```

12. STATEMENTS

The body of a program, procedure, or function contains statements. Statements denote actions that the program can execute. This chapter first discusses the syntax of statements and then separates and describes two categories of statements: simple statements and structured statements. A simple statement has no parts that are themselves other statements; a structured statement consists of two or more other statements. Table 12-1 lists the statements in each category in MS-Pascal.

Table 12-1. MS-Pascal Statements

SIMPLE	STRUCTURED
Assignment (:=)	Compound
Procedure	IF/THEN/ELSE
GOTO	CASE
BREAK	FOR
CYCLE	WHILE
RETURN	REPEAT
Empty	WITH

12.1 THE SYNTAX OF PASCAL STATEMENTS

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 in the following sections.

12.1.1 LABELS

Any statement referred to by a GOTO statement must have a label. A label at the standard level is one or more digits; leading zeros are ignored. Constant

identifiers, expressions, and nondecimal notation cannot serve as labels. All labels must be declared in a LABEL section. At the extend level, a label can also be an identifier.

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, or FOR. At the extend level, a BREAK or CYCLE statement can also 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, and 123 is label:

```
321:  123:  IF LOOP THEN GOTO 123
```

12.1.2 SEPARATING STATEMENTS

Semicolons separate statements. Semicolons do not terminate statements. However, since Pascal permits the empty statement, using the semicolon as if it were a statement terminator is rarely disastrous.

Example showing semicolon to separate statements:

```
BEGIN
   10: WRITELN;
   A := 2 + 3;
   GOTO 10
```

```
END
```

A common error is to terminate the THEN clause in an IF/THEN/ELSE statement with a semicolon. Thus, the following generates a warning message:

```
IF A = 2 THEN WRITELN;
ELSE A = 3
```

Another common error is to put a semicolon after the DO in a WHILE or FOR statement:

```
FOR I := 1 TO 10 DO;
BEGIN
  A[I] := I;
  B[I] := 10 - I;
END;
```

This example, as written, "executes" an empty statement ten times, then executes 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.

12.1.3 THE RESERVED WORDS BEGIN AND END

Whenever you want a program to execute a group of statements, instead of a single simple statement, you may enclose the block with the reserved words BEGIN and END.

For example, the following group of statements between BEGIN and END are executed if the condition in the IF statement is TRUE:

```
IF (MAX > 10) THEN
BEGIN
  MAX = 10;
```

```
    MIN = 0;
    WRITELN (MAX,MIN)
  END;
  WRITELN ('done')
```

At the extend level, you may substitute a pair of square brackets for the pair of keywords BEGIN and END.

12.2 SIMPLE STATEMENTS

A simple statement is one in which no part constitutes another statement. Simple statements in standard Pascal are:

1. The assignment statement

2. The procedure statement

3. The GOTO statement

4. The empty statement

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.

The extend level in MS-Pascal adds three simple statements: BREAK, CYCLE, and RETURN.

12.2.1 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 adjacent colon and equal sign characters (:=).

Examples of assignment statements:

```
A := B

A[I] := 12 * 4 + (B * C)

X : = Y
{Illegal.  Colon (:) and equal}
{sign (=) must be adjacent.}

A + 2 := B
{Illegal.  A + 2 is not a variable.}

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.

An assignment to a nonlocal variable (including a function return) puts an equal sign (=) or percent sign (%) in the G column of the listing file. (See Section 17.5 for more information about these and other symbols used in the listing.)

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 can occur either within the actual body of the function or in the body of a procedure or function nested within it.

If the range checking switch is on, an assignment to a set, subrange, or LSTRING variable may imply a run-time call to the error checking code.

According to the MS-Pascal optimizer, each section of code without a label or other point that could

receive control is 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 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 for "aliases," that is, a single variable with two identifiers, perhaps one as a global variable and one as a reference parameter.

12.2.2 PROCEDURE STATEMENTS

A procedure statement executes the procedure denoted by the procedure identifier.

For example, assume you have defined the procedure DO_IT:

```
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.

MS-Pascal includes a large number of predeclared procedures. See Chapter 14 for complete information. One of the predeclared procedures is ASSIGN. You need not declare it in order to use it.

```
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. For a discussion of formal and reference parameters, see Section 13.4.

12.2.3 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.

Two 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 are permitted from one branch of an IF or CASE statement to another.

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.

Examples of GOTO statements, both legal and illegal:

```
PROGRAM LABEL_EXAMPLES;
LABEL 1, 2, 3, 4;
```

```
  PROCEDURE ONE;
  LABEL 11, 12, 13;

    PROCEDURE IN_ONE;
    LABEL 21;
    {Outer level GOTOs cannot jump in to 21.}

    BEGIN
      IF TUESDAY THEN GOTO 1
      ELSE GOTO 11;
      {1 and 11 are both legal outer level labels.}
      21: WRITE ('IN_ONE')
    END;

  BEGIN   {Procedure one}
    IF RAINING THEN GOTO 1 ELSE GOTO 11;
    {That was legal.}
    11: GOTO 21;
    {Illegal.  Cannot jump into inner level}
    {procedures.}
  END;

  PROCEDURE TWO;
  BEGIN
    GOTO 11
    {Illegal.  Cannot jump into different procedure}
    {at same level}
  END;

BEGIN   {Main level}
  IF SEATTLE
  THEN
    BEGIN BEGIN
      GOTO 2;
      {OK to go to 2 at program level.}

      4: WRITE ('here');
    END END
  ELSE GOTO 4;
  {OK to jump into THEN clause.}
  2: GOTO 3;
```

```
{Illegal. Cannot jump into REPEAT statement.}
REPEAT
    WHILE MS_BYRON DO
      3: GOTO 2
      {OK to jump out of loops.}
UNTIL DATE;
1: GOTO 11;
{Illegal.  Cannot jump into procedure from program.}

END.
```

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.

See Section 17.5 for details about the listing file.

12.2.4 THE BREAK, CYCLE, AND RETURN STATEMENTS

At the extend level, 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 (+) or an asterisk (*) for a BREAK statement, a minus (-) or asterisk (*) for a CYCLE statement, and an asterisk (*) for a RETURN statement. (See Section 17.5 for information about the listing file.)

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;
```

12.3 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. You can use the C column to search for a missing or extra END in a program.

12.3.1 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
   {Semicolon not needed here.}
END

BEGIN
   OPEN_DOOR;
```

```
    LET_EM_IN;
    CLOSE_DOOR;
    {Semicolon signifies empty statement.}
END
```

All MS-Pascal conditional and repetitive control structures (except REPEAT) operate on a single statement, not on multiple statements with ending delimiters. In this context, BEGIN and END serve as punctuation, like semicolon, colon, or parentheses. If you prefer, you can 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 cannot 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 := 0];

WHILE P.N <> NIL DO
   [Q := P; P := P.N; DISPOSE (Q)];

FUNCTION R2 (R: REAL): REAL;
   [R2 := R * 2]
   {Illegal.}
```

12.3.2 CONDITIONAL STATEMENTS

A conditional statement selects for execution only one of its component statements. The conditional statements are the IF and CASE statements. Use the IF statement for one or two conditions, the CASE statement for multiple conditions.

12.3.2.1 The IF Statement

The IF statement allows for conditional execution of a statement. If the Boolean expression following IF is true, the statement following THEN is executed. If the Boolean expression following IF is false, the statement following ELSE, if present, is executed.

Examples of IF statements:

```
IF I > 0 THEN I := I - 1
{No semicolon here.}
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 Boolean expression following an IF may include the sequential control operators described in Section 12.3.5.

12.3.2.2 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.  In this instance,}
{they are all of the values assumable by the}
{enumerated variable, OPERATOR.}

CASE NEXTCH OF
  'A'..'Z', '_'        : IDENTIFIER;
  '+', '-', '*', '/'   : OPERATOR;
  {Commas separate CASE constants}
  {and ranges of CASE constants.}
  OTHERWISE
    WRITE ('Unknown Character')
    {I.e., if any other 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. At the extend level, 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 extend level also allows the CASE statement to end with an OTHERWISE clause. The OTHERWISE clause contains additional statements to be executed in the event the CASE index value is not in the given set of CASE constant values. One of the following two things happens if the CASE index value is not in the set and no OTHERWISE clause is present:

1. If the range checking switch is on, a run-time error is generated.

2. If the range checking switch is off, the result is undefined (and may be catastrophic).

In MS-Pascal, control does not automatically pass to the next executable statement as in UCSD Pascal and some other languages. If you want this effect, include an empty OTHERWISE clause.

A semicolon may appear after the final statement in the list, but is not required. The compiler skips over a colon after an OTHERWISE and issues a warning.

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.

12.3.3 REPETITION STATEMENTS

Repetition statements specify repeated execution of

a statement. In standard Pascal, these include the WHILE, REPEAT, and FOR statements.

At the extend level in MS-Pascal, there are two additional statements, BREAK and CYCLE, for leaving or restarting the statements being repeated. These statements are functionally equivalent to a GOTO but are easier to use.

12.3.3.1 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 Boolean expression in a WHILE statement can include the sequential control operators described in Section 12.3.5. Use WHILE if it is possible that no iterations of the loop may be necessary; use REPEAT where you expect that at least one iteration of the loop is required.

12.3.3.2 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;
```

The Boolean expression in a REPEAT statement may include the sequential control operators described in Section 12.3.5. Use the REPEAT statement to execute statements (not just a single statement) 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.

12.3.3.3 The FOR Statement

The FOR statement tells 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 10 DO
{I is the control variable.}
  SUM := SUM + VICTORVECTOR [I]

FOR CH := 'Z' DOWNTO 'A' DO
{CH is the control variable.}
  WRITE (CH)
```

You may also use a FOR statement to step through the values of a set, as shown:

```
FOR TINT :=
   LOWER (SHADES) TO UPPER (SHADES) DO
      IF TINT IN SHADES
```

```
THEN PAINT_AREA (TINT);
```

The ISO standard gives explicit rules regarding the control variable in FOR statements:

1. The control variable 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.

   Howvever, at the extend level of MS-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. Using a program level variable is an ISO error not caught.

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 intitial 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, can 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.

In standard Pascal, 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.

Also, a control variable with the STATIC attribute may be more efficient then one that is not.

At the extend level in MS-Pascal, you can 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 in 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
```

12.3.3.4 The BREAK And CYCLE Statements

In theory, a program using the MS-Pascal extend level BREAK and CYCLE statements does not need to use any GOTO statements.

Both the BREAK and CYCLE statements have 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. Since at the extend level, you can use identifier labels, a suggested practice is to 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;
```

12.3.4 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 listed 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.

In MS-Pascal, 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. Avoid assignments to the WITH record itself in programs intended to be portable.

12.3.5 SEQUENTIAL CONTROL

To increase execution speed or guarantee 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. Two extend level operators in MS-Pascal provide for such tests:

1. AND THEN

   X AND THEN Y is false if X is false; Y is evaluated only if X is true.

2. OR ELSE

3. X OR ELSE Y is true if X is true; Y is evaluated only if X is false.

If you use several sequential control operators, the compiler evaluates them strictly from left to right.

You can include these operators only in the Boolean expression of an IF, WHILE, or UNTIL clause; they cannot 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
```

13. INTRODUCTION TO PROCEDURES AND FUNCTIONS

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 three-part structure includes a heading, declarations, and a body.

Example of a procedure declaration:

```
{Heading}
PROCEDURE MODEL (I: INTEGER; R: REAL);

{Beginning of declaration section}
LABEL 123;
CONST ATOP = 199;
TYPE  INDEX = 0..ATOP;
VAR   ARAY: ARRAY [INDEX] OF REAL; J: INDEX;

{Function declaration}
FUNCTION FONE (RX: REAL): REAL;
BEGIN
   FONE := RX * I
END;

{Procedure declaration}
PROCEDURE FOUT (RY: REAL);
BEGIN
   WRITE ('Output is ', RY)
END;
```

```
{Body of procedure MODEL}
BEGIN
  FOR J := 0 TO ATOP DO
    IF GLOBALVAR THEN
    {Activation of procedure FOUT with}
    {value returned by function FONE.}
        FOUT (FONE (R + ARAY [J]))
      ELSE GOTO 123;
  123:  WRITELN ('Done');
END;
```

The declarations and body together are called the block.

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.

13.1 PROCEDURES

The preceding 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:

1. LABEL

2. CONST

3. TYPE

4. VAR

5. Procedures and functions

At the extend level, you can have any number of LABEL, CONST, TYPE, VAR, and VALUE sections, as well as procedure and function declarations, in any order. Although data declarations (CONST, TYPE, VAR, VALUE) can be intermixed with procedure and function declarations, it is usually clearer to give all data declarations first.

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, is an MS-Pascal extension that 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.

13.2 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 run-time, 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 extend 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, use the function RESULT, which takes the function identifier as a parameter and is available at the extend level.

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 these:

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. As mentioned in Section 10.1 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 shown:

```
TYPE IREC = RECORD I: INTEGER END;
 FUNCTION SUM (A, B: INTEGER): IREC;
 {Return sum of A and B.}
 BEGIN
   IF TUESDAY THEN
   BEGIN                {On Tuesdays, we recurse!}
     IF B = 0 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                  {Use function's}
     WITH SUM            {local variable.}
       DO I := A + B;    {I is local variable.}
 END
```

13.3 ATTRIBUTES AND DIRECTIVES

An attribute gives additional information about a procedure or function. Attributes are available at the extend level of MS-Pascal. They are placed after the heading, enclosed in brackets and separated by commas. Available attributes include ORIGIN, PUBLIC, FORTRAN, PURE, and INTERRUPT.

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. Directives are available in standard Pascal. EXTERN and FORWARD are the only directives available. EXTERN can be used only with procedures or functions directly nested in a program, module, implementation, or interface. This restriction prevents access to nonlocal stack variables.

Table 13-1 displays the attributes and directives that apply to procedures and functions. Sections 13.3.1 through 13.3.7 describe these attributes in

detail.

Table 13-1. Attributes and Directives for 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 can be accessed by other loaded modules.
ORIGIN	An attribute. Tells the compiler where the code for an EXTERN procedure or function resides.
FORTRAN	An attribute. Specifies a calling sequence for compatibility with MS- FORTRAN.
INTERRUPT	An attribute. Gives a procedure a special calling sequence that saves program status on the stack.
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 can 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. A given procedure or function can have only one calling sequence attribute (either FORTRAN or INTERRUPT, but not both).

4. 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 can give the EXTERN directive in an implementation by declaring all EXTERN procedures and functions first; you cannot use ORIGIN in either the interface or implementation of a unit.

In a module, you can 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 can 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 MS-Pascal routines with other routines in MS-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 the program. FORWARD routines are fully declared (have a block) later in the same compiland. Both directives are available at the standard level of MS-Pascal. The keyword EXTERNAL is a synonym for EXTERN.

The PURE attribute applies only to functions, not to procedures. Conversely, INTERRUPT applies only to procedures, not to functions. PURE is the only attribute that can be used in nested functions.

13.3.1 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 make a FORWARD declaration by specifying a procedure or function heading, followed by the directive FORWARD. Later, you 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:

```
{Declaration of ALPHA, with parameter}
 {list and attributes}
FUNCTION ALPHA (Q, R: REAL): REAL [PUBLIC];
  FORWARD;
```

```
{Call for ALPHA}
PROCEDURE BETA (VAR S, T: REAL);
BEGIN
  T := ALPHA (S, 3.14)
END;

{Actual declaration of ALPHA,}
{without parameter list}
FUNCTION ALPHA;
BEGIN
  ALPHA := (Q + R);
  IF R < 0.0 THEN BETA (3.14,ALPHA);
END;
```

13.3.2 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. As with variables, the keyword EXTERNAL is a synonym for EXTERN.

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. You can also link MS-Pascal routines by linking separately compiled units (see Chapter 16).

Examples of procedure and function headings declared with the EXTERN directive:

```
FUNCTION POWER (X, Y: REAL): REAL; EXTERN;
  PROCEDURE ACCESS (KEY: KTYP) [ORIGIN SYSB+4];
EXTERN;
```

In these examples, the function POWER is declared EXTERN, as is the procedure ACCESS. Both are implemented in external compilands. ACCESS also has the ORIGIN attribute, which is discussed, in Section 13.3.4.

You cannot declare a procedure or function EXTERN if you have previously declared it FORWARD.

13.3.3 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 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. See Appendix A in your MS- Pascal User's Guide for specific information on limitations that your version of the compiler and linker may set on identifiers. 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. A higher level way to link MS-Pascal routines is by linking separately compiled

units. (see Chapter 16 for details).

Examples of procedures and functions declared PUBLIC:

```
FUNCTION POWER (X, Y: REAL): REAL [PUBLIC];
 {The function POWER is available to other}
 {modules because it has been declared PUBLIC.}
 BEGIN
    .
    .
 END;

 PROCEDURE ACCESS (KEY: KTYP) [ORIGIN SYSB+4,
   PUBLIC];

 BEGIN
    .
    .
 END;
 {Illegal since ORIGIN must also be EXTERN.}
```

13.3.4 THE ORIGIN ATTRIBUTE

The ORIGIN attribute must be used with the EXTERN directive; ORIGIN tells the compiler where the procedure or function can be found directly, so the linker does not require a corresponding PUBLIC identifier.

Examples of procedures and functions given the ORIGIN attribute:

```
PROCEDURE OPSYS [ORIGIN 8, FORTRAN]; EXTERN;

FUNCTION A_TO_D (C: SINT): SINT [ORIGIN #100];
EXTERN;
```

In the first example, the procedure OPSYS begins at the absolute decimal address 8, has the FORTRAN calling sequence (described in Section 13.3.5), and is declared EXTERN. In the second 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.

As with ORIGIN variables, the compiler uses the address to find the code and gives no directives to the linker. This permits, for example, calling routines at fixed addresses in ROM. In simple cases, it can substitute for a linking loader.

Remember that ORIGIN always implies EXTERN. Thus, procedures or functions that have previously been declared FORWARD cannot be declared with the ORIGIN attribute. Also, you cannot give ORIGIN as an attribute after the module heading. Currently, you cannot use the ORIGIN attribute with a constituent of a unit, either in an interface or in an implementation.

As with variables, the origin can be a segmented address. On segmented machines, a nonsegmented procedural origin assumes the current code segment with the offset given with the attribute; this form has no obvious uses.

13.3.5 THE FORTRAN ATTRIBUTE

The FORTRAN attribute applies both to procedures and functions (but not to variables). Instead of the usual Pascal calling sequence, it specifies a calling sequence that is compatible with the MS-FORTRAN compiler on your machine.

This attribute lets you call an MS-Pascal procedure or function from MS-FORTRAN programs and, conversely, external FORTRAN subroutines or MS-FORTRAN functions from an MS-Pascal program.

Example of a procedure with the FORTRAN attribute:

```
PROCEDURE DELTA (I, J: INTEGER) [FORTRAN];
FORWARD;
```

Any procedure or function with the FORTRAN attribute must be nested directly within a program or implementation.

In a 16-bit environment, MS-Pascal uses the same calling sequence as the compilers for MS-(@) FORTRAN, MS-(@) BASIC, and MS-(@) COBOL. Thus, there is no need to give the FORTRAN attribute; if you do, it is ignored by the MS-Pascal . See Appendix A. in your MS- Pascal User's Guide for details on the MS-Pascal calling sequence.

13.3.6 THE INTERRUPT ATTRIBUTE

The INTERRUPT attribute applies only to procedures, not to functions or variables. It gives a procedure a special calling sequence that saves program status on the stack, which in turn allows a hardware interrupt to be processed, status restored, and control returned to the program, all without affecting the current state of the program.

Example of a procedure with the INTERRUPT attribute:

```
PROCEDURE INCHAR [INTERRUPT];
```

Because procedures with the INTERRUPT attribute are intended to be invoked by hardware interrupts, you cannot invoke them with a procedure statement. An INTERRUPT procedure can be invoked only when the interrupt associated with it occurs. Furthermore, INTERRUPT procedures take no parameters.

Declaring a procedure with the INTERRUPT attribute ensures that the procedure conforms to the constraints of an interrupt handler in which:

1. A special calling sequence saves all status on

the stack.

2. The status saved includes machine registers and flags, plus any special global compiler data such as the frame pointer.

3. The saved status is restored upon exit from the procedure.

All INTERRUPT procedures must be nested directly within a compiland.

Interrupts are not automatically vectored to INTERRUPT procedures; further, insofar as possible on the target machine, interrupts are neither enabled or disabled by an INTERRUPT procedure. Interrupt vectoring and enabling are too machine-dependent to be included in a machine-independent language like MS-Pascal.

However, MS-Pascal does provide the VECTIN library procedure, which takes an interrupt level and an interrupt procedure as parameters and sets the interrupt vector in a machine-dependent way. Similarly, the library procedures ENABIN and DISBIN, respectively, enable and disable interrupts in a machine-dependent way. See Chapter 14 for more information on these routines. See also Appendix A, in the MS- Pascal User's Guide for information about the implementation of these routines under your operating system.

An INTERRUPT procedure should usually return normally, in order to continue processing in the interrupted routine. This means the following:

1. You should not execute a GOTO that leaves an INTERRUPT procedure.

2. All debug checking should be turned off ($DEBUG-, $ENTRY-, and $RUNTIME-).

3. Stack overflow cannot be checked even if $STACKCK is on.

The use of INTERRUPT procedures introduces re-entrancy into MS-Pascal code: generated code is re-entrant, as is the run-time system (except for the heap unit and, in most operating systems, portions of the file unit).

Some critical sections in the run-time system are protected by semaphores that generate a run-time error if such a critical section is locked. For example, if the heap allocator is executing when an interrupt occurs and the INTERRUPT procedure tries to allocate a block from the heap, the structure of the heap could become invalid. This condition causes a run-time error.

However, in most cases, the file system is not protected by a semaphore. Therefore, it is safest to avoid performing any I/O within the INTERRUPT procedure. Alternatively, you can avoid most problems with I/O in an INTERRUPT procedure by not opening or closing any files (not declaring any local file variables or creating files on the heap) and by not performing input or output with any file that might be performing I/O when the interrupt occurs.

13.3.7 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];
```

For further 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 generates code to compute I*10 only once. However, FOO, if it is not declared PURE, may modify I so that I*10 must be recomputed after the call to FOO.

Functions are not considered PURE unless given the attribute explicitly. The compiler checks to see that a PURE function does not do any of the following:

- Assign to a nonlocal variable
- Have any VAR or VARS parameters (CONST and CONST parameters are permitted)
- Call any functions that are not PURE
- Modify global variables

Although the following additional restrictions are not checked, a PURE function should also not:

- Use the value of a global variable.
- Modify the referents of references passed by value (e.g., pointer or address type referents).
- 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);
```

13.4 PROCEDURE AND FUNCTION PARAMETERS

Procedures and functions may take three different types of parameters:

1. Value parameters

2. Reference parameters

3. Procedural and functional parameters

Each of these is discussed separately, in the order listed, in the following paragraphs.

The discussion mentions both formal and actual 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.

MS-Pascal has the following parameter features at the extend 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.

13.4.1 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 declaration }
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 this particular function invocation, Y, ADD(...), and (Z * 4) are the expressions that make up the actual parameters. In this example, these expressions must all evaluate to the type REAL. (The example also recursively calls the function ADD.)

The actual parameter expression must be assignment compatible with the type of the formal parameter.

Passing structured types by value is permitted; 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 run-time error check if the range checking switch 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.

13.4.2 REFERENCE PARAMETERS

When a reference parameter is passed at the standard level of MS-Pascal, 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. For target processors with segmentation support, 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.}
  {They denote variables, not values.}
    .
    .
CHANGE_VARS (X, Y, Z);
```

In this 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 Section 17.5, for information about significance of these characters in the G column of the listing).

Neither of the following may 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 or PORT 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.

On a segmented machine, 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 tell the compiler to use a segmented address containing both segment register and offset values. The extend level includes the parameter prefix VARS instead of VAR:

```
PROCEDURE CONCATS (VARS T, S: STRING);
```

You may only use VARS as a data parameter in procedures and functions, not in the declaration section of programs, procedures, and functions. VARS and CONSTS parameters are provided chiefly to maintain compatibility with machines that have two different size address spaces. These parameters are not necessary for a machine with a single size address space. On such machines, the reserved words VARS and CONSTS are equivalent to VAR and CONST.

13.4.2.1 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 [0..*] OF REAL;

PROCEDURE SUMRS (VAR X: REALS; CONST X: REALS);
BEGIN
  .
  .
END;
```

For more information, see Sections 6.2, 6.2.1, and 6.2.2.

13.4.2.2 Constant And Segment Parameters

At the extend 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);
```

On a segmented machine, 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 tell the compiler to use a segmented address that contains both segment register and offset values. The extend 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);
```

A CONSTS parameter can be used as a data parameter only 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 shown:

```
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;
```

13.4.3 PROCEDURAL AND FUNCTIONAL PARAMETERS

Procedural parameters can be used in the following circumstances:

1. In numerical analysis

2. In calling some library routines

3. In special applications

In numerical analysis, you might pass a function to a procedure or function that finds an integral between limits, a maximum or minimum value, and so on. Some interesting algorithms in areas such as parsing and artificial intelligence also use procedural 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 and 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) = 0
 THEN
    SOCIETY := SAFE;
 IF DOOR_STATUS
    (OPEN_DOOR_WIDE, BARN, SLOW, OPEN) = 0
 THEN
    COWS_ARE_OUT := TRUE;
 IF DOOR_STATUS
```

```
    (LEAVE_AJAR, DOG_HOUSE, SLOW, OPEN) = 0
  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.

In MS-Pascal, you cannot pass predeclared procedures and functions compiled as inline code; you can pass them only 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, however, can be passed. 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 of MS-Pascal:

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 extend and system levels of MS-Pascal:

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 what happens in this 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 0, 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.

14. AVAILABLE PROCEDURES AND FUNCTIONS

All versions of Pascal predeclare a large number of common procedures and functions. You do not have to declare these procedures and functions in a program. Since they are defined in a scope "outside" the program, you may redefine these identifiers.

To promote portability, MS-Pascal makes some of the predeclared procedures and functions available only at the extend or at the system level. MS-Pascal also includes some useful library procedures and functions that you must declare EXTERN in order to use.

MS-Pascal implements three kinds of procedures and functions:

1. Some are predeclared, and the compiler translates them into other calls or special generated code (these you cannot pass as parameters).

2. Some are predeclared but you call them normally (except for a name change).

3. Some are not predeclared but available as part of the MS-Pascal run-time library (these you must declare explicitly).

However, it is more useful when discussing these procedures and functions to categorize them by what they do rather than how they are implemented. Table 14-1 shows this categorization.

Table 14.1. Categories of Available Procedures and Functions

CATAGORY	PURPOSE
File system	Operate on files of different modes and structures.
Dynamic allocation	Dynamically allocate and deallocate data structures on the heap at run-time.
Data conversion	Convert data from one type to conversion another.
Arithmetic	Perform common transcendental and other numeric functions.
Extend level intrinsics	Provide additional procedures and functions at the extend level of MS-Pascal.
System level intrinsics	Provide additional procedures and functions at the system level of MS-Pascal.
String intrinsics	Operate on STRING and LSTRING type data.
Library	Available in the MS-Pascal run-time library. They are not predeclared; you must declare them with the EXTERN directive.

14.1 CATEGORIES OF AVAILABLE PROCEDURES AND FUNCTIONS

This section describes each of the categories listed in Table 14-1 and lists the procedures and functions included in each category. See Section 14.2 for an alphabetical directory of all of the available procedures and functions.

14.1.1 FILE SYSTEM PROCEDURES AND FUNCTIONS

The MS-Pascal file system supports a variety of procedures and functions that operate on files of different modes and structures. These procedures and functions fall into three categories, as shown in Table 14-2.

Table 14.2. File System Procedures and Functions

CATEGORY	PROCEDURES	FUNCTIONS
Primitive	GET	EOF
	PAGE	EOLN
	PUT	
	RESET	
	REWRITE	
Textfile I/O	READ	
	READLN	
	WRITE	
	WRITELN	
Extend Level I/O	ASSIGN	
	CLOSE	
	DISCARD	
	READSET	
	READFN	
	SEEK	

For details on each of these procedures and functions, see Chapter 15.

14.1.2 DYNAMIC ALLOCATION PROCEDURES

Two procedures, NEW and DISPOSE, allow dynamic allocation and deallocation of data structures at run-time. NEW allocates a variable in the heap, and DISPOSE releases it.

14.1.3 DATA CONVERSION PROCEDURES AND FUNCTIONS

Use the following procedures and functions to convert data from one type to another:

CHR	PACK	TRUNC
FLOAT	PRED	TRUNC4
FLOAT4	ROUND	UNPACK
ODD	ROUND4	WRD
ORD	SUCC	

Four of these convert any ordinal type to a particular ordinal type:

CHR (ordinal) to CHAR
ODD (ordinal) to BOOLEAN
ORD (ordinal) to INTEGER
WRD (ordinal) to WORD

PRED and SUCC also operate on ordinal types.

Six of the conversion procedures and functions convert between INTEGER or INTEGER4 and REAL:

FLOAT	converts	INTEGER	to	REAL
FLOAT4	converts	INTEGER4	to	REAL
ROUND	converts	REAL	to	INTEGER
ROUND4	converts	REAL	to	INTEGER4
TRUNC	converts	REAL	to	INTEGER
TRUNC4	converts	REAL	to	INTEGER4

PACK and UNPACK transfer components between packed and unpacked arrays.

14.1.4 ARITHMETIC FUNCTIONS

All arithmetic functions take a CONSTS parameter of type REAL4 or REAL8, or a type compatible with INTEGER (labeled "numeric" in the directory). 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 run-time 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 run-time error message. If the switch is off, the result of an error is undefined.

Table 14-3 lists the arithmetic function available, along with the routines called depending on whether single or double precision is required.

Table 14-3: Predeclared Arithmetic Functions

NAME	OPERATION	REAL4	REAL8
ABS	Absolute value	(inline)	(inline)
ARCTAN	Arctangent	ATSRQQ	ATDRQQ
COS	Cosine	CNSRQQ	CNDRQQ
EXP	Exponential	EXSRQQ	EXDRQQ
LN	Natural log	LNSRQQ	LNDRQQ
SIN	Sine	SNSRQQ	SNDRQQ
SQR	Square	(inline)	(inline)
SQRT	Square root	SRSRQQ	SRDRQQ

The MS-FORTRAN run-time library provides several additional REAL4 and REAL8 functions, as shown in Table 14.4. If you use them, you must declare them with the EXTERN directive.

Table 14-4: REAL Functions from the MS-FORTRAN Run-time Library

OPERATION	REAL4	REAL8
Arccosine	ACSRQQ	ACDRQQ
Integral trunc	AISRQQ	AIDRQQ
Integral round	ANSRQQ	ANDRQQ
Arcsine	ASSRQQ	ASDRQQ
Arctangent A/B	A2SRQQ	A2DRQQ
Hyperbolic cosine	CHSRQQ	CHDRQQ
Decimal log	LDSRQQ	LDDRQQ
Modulo	MDSRQQ	MDDRQQ
Minimum	MNSRQQ	MNDRQQ
Maximum	MXSRQQ	MXDRQQ
Power (REAL8**INTG4)		PIDRQQ
Power (REAL4**INTG4)	PISRQQ	
Power (REAL ** REAL)	PRSRQQ	PRDRQQ
Hyperbolic sine	SHSRQQ	SHDRQQ
Hyperbolic tangent	THSRQQ	THDRQQ
Tangent	TNSRQQ	TNDRQQ

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 follow:

```
SIGN (X)      is    ORD (X > Ø) - ORD (X < Ø)
POWER (X, Y)  is    EXP (Y * LN (X))
```

You can also write your own functions in MS-Pascal to do the same thing. Defining functions like these is a good opportunity to use the PURE attribute (to obtain more efficient code). 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;
```

14.1.5 EXTEND LEVEL INTRINSICS

At the extend level of MS-Pascal, the following intrinsic procedures and functions are available:

ABORT	EVAL	LOWORD
BYLONG	HIBYTE	RESULT
BYWORD	HIWORD	SIZEOF
DECODE	LOBYTE	UPPER
ENCODE	LOWER	

Several of these are used to compose and decompose one-byte, two-byte, and four-byte items: HIBYTE, LOBYTE, BYWORD, HIWORD, LOWORD, and BYLONG.

ENCODE and DECODE convert between internal and string forms of variables. ABORT invokes a run-time error.

The others, EVAL, LOWER, UPPER, RESULT, and SIZEOF, are used in special situations (described for each function in Section 14.2.

14.1.6 SYSTEM LEVEL INTRINSICS

Several additional intrinsic procedures and functions are available at the system level:

FILLC	MOVESL
FILLSC	MOVESR
MOVEL	RETYPE
MOVER	

The MOVE and FILL procedures perform low-level operations on byte strings. RETYPE changes the type of an expression arbitrarily.

14.1.7 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:

Table 14-5: String Procedures and Functions

NAME	PARAMETER
CONCAT	STRING
DELETE	STRING
INSERT	STRING
COPYLST	STRING
COPYSTR	STRING or LSTRING
POSITN	STRING or LSTRING
SCANEQ	STRING or LSTRING
SCANNE	STRING or LSTRING

14.1.8 LIBRARY PROCEDURES AND FUNCTIONS

The following routines are not predeclared, but are available to you in the MS-Pascal run-time library. You must declare them, with the EXTERN directive, before using them in a program.

Initialization and Termination Routines

BEGOQQ and ENDOQQ are called during initialization and termination, respectively. You might use them to invoke a debugger or to write customized messages, such as the time of execution, to the terminal screen. BEGXQQ can be called to restart a program and ENDXQQ to terminate it.

Heap Management Routines

Heap management routines complement the standard NEW and DISPOSE procedures and include:

ALLHQQ FREECT MARKAS MEMAVL RELEAS

Interrupt Routines

These routines handle interrupt processing, although the actual effect varies with the target machine:

ENABIN DISABIN VECTIN

Terminal I/O Routines

The following routines support direct input to and output from your terminal:

GTYUQQ PTYUQQ PLYUQQ

Semaphore Routines

The two procedures, LOCKED and UNLOCK, provide a binary semaphore capability. You can use them to ensure exclusive access of a resource in a concurrent system.

No-Overflow Arithmetic Functions

These functions implement 16-bit and 32-bit modulo arithmetic. Overflow or carry is returned, instead of invoking a run-time error.

LADDOK LMULOK SADDOK SMULOK UADDOK UMULOK

Clock Routines

These provide operating system clock information:

TIME DATE TICS

14.2 DIRECTORY OF FUNCTIONS AND PROCEDURES

This section contains a lists all available procedures and functions, both those that are predeclared and those library routines that may be used if declared EXTERN. Each entry includes the heading, the category to which the operation belongs, and a description of what the procedure or function does. Notes and examples are included as appropriate. The headings given are the same for both REAL4 or REAL8, unless specifically stated otherwise.

PROCEDURE ABORT (CONST STRING, WORD, WORD);

An extend level intrinsic procedure. Halts program execution in the same way as an internal run-time 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 H 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 report 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 call only other $RUNTIME routines.

FUNCTION ABS (X: NUMERIC): NUMERIC;

An arithmetic 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 ACSRQQ (CONSTS A: REAL4): REAL4;
FUNCTION ACDRQQ (CONSTS A: REAL8): REAL8;

Arithmetic functions. Return the arccosine of A. Both A and the return value are of type REAL4 or REAL8, as shown. These functions are from the MS-FORTRAN run-time library and must be declared EXTERN before use.

FUNCTION AISRQQ (CONSTS A: REAL4): REAL4;
FUNCTION AIDRQQ (CONSTS A: REAL8): REAL8;

Arithmetic functions. Return the integral part of A, truncated toward zero. Both A and the return value are of type REAL4 or REAL8, as shown. These functions are from the MS-FORTRAN run-time library and must be declared EXTERN before use.

FUNCTION ALLHQQ (SIZE: WORD): WORD;
A library routine (heap management function). Returns zero if the heap is full, 1 if the heap structure is in error, or MAXWORD if the allocator has been interrupted. Otherwise, it returns the pointer value for an allocated variable with the size requested.

Generally, you use ALLHQQ with the RETYPE function. For example:

```
P_VAR := RETYPE (P_TYPE, ALLHQQ (28));
{RETYPE converts the value returned by}
{ALLHQQ (28) to the type P_TYPE.}
{This value is assigned to P_VAR.}

IF  WRD (P_VAR) < 2 THEN GO_ABORT;
{PVAR is then checked for a heap}
{full or heap structure error.}
```

FUNCTION ANSRQQ (CONSTS A: REAL4): REAL4;
FUNCTION ANDRQQ (CONSTS A: REAL8): REAL8;

Arithmetic functions. Like AISRQQ and AIDRQQ, return the truncated integral part of A, but round away from zero. Both A and the return value are of type REAL4 or REAL8, as shown. These functions are from the MS-FORTRAN run-time library and must be declared EXTERN before use.

FUNCTION ARCTAN (X: REAL): REAL;

An arithmetic function. Returns the arctangent of X in radians. Both X and the return value are of type REAL. To force a particular precision, declare ATSRQQ (CONSTS REAL4) and/or ATDRQQ (CONSTS REAL8) and use them instead.

FUNCTION ASSRQQ (CONSTS A: REAL4): REAL4;
FUNCTION ASDRQQ (CONSTS A: REAL8): REAL8;

Arithmetic functions. Return the arcsine of A. Both A and the return value are of type REAL4 or REAL8, as shown. These functions are from the MS-FORTRAN run-time library and must be declared EXTERN before use.

PROCEDURE ASSIGN (VAR F; CONSTS N: STRING);

A file system procedure (extend level I/O). Assigns an operating system filename in a STRING (or LSTRING) to a file F.

See Section 15.3.1 for a description of ASSIGN.

FUNCTION A2SRQQ (A, B: REAL4): REAL4;
FUNCTION ASDRQQ (A, B: REAL8): REAL8;

Arithmetic functions. Return the arctangent of (A/B). Both A and B, as well as the return value, are of type REAL4 or REAL8, as shown. These functions are from the MS-FORTRAN run-time library and must be declared EXTERN before use.

PROCEDURE BEGOQQ;

A library routine (initialization). BEGOQQ 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.

See also PROCEDURE ENDOQQ.

PROCEDURE BEGXQQ;

A library routine (initialization). 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. Resets the stack and the heap.

2. Initializes the file system.

3. Calls BEGOQQ.

4. Calls the program body.

BEGXQQ can be useful for restarting after a catastrophic error in a ROM-based system. However, invoking this procedure to restart a program does not close 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.

FUNCTION BYLONG (INTEGER-WORD, INTEGER-WORD): INTEGER4;

An extend level intrinsic 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;

An extend level intrinsic function. Converts bytes (or the LOBYTEs of INTEGERs or WORDs) to a WORD value. 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 CHR (X: ORDINAL): CHAR;

A data conversion function. Converts any ordinal type to CHAR. The ASCII code for the result is ORD (X). This is an extension to the ISO standard, which requires X to be of type INTEGER. An error occurs if ORD (X) > 255 or ORD (X) < 0. However, the error is caught only if the range checking switch is on.

FUNCTION CHSRQQ (CONSTS A: REAL4): REAL4;
FUNCTION CHDRQQ (CONSTS A: REAL8): REAL8;

Arithmetic functions. Return the hyperbolic cosine of A. Both A and the return value are of type REAL4 or REAL8, as shown. These functions are from the MS-FORTRAN run-time library and must be declared EXTERN before use.

PROCEDURE CLOSE (VAR F);

A file system procedure (extend level I/O). Performs an operating system close on a file, ensuring that the file access is terminated correctly.

See Section 15.3.1 for a description of CLOSE.

PROCEDURE CONCAT (VARS D: LSTRING; CONSTS S: STRING);

A string intrinsic 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);

A string intrinsic 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; VARS D: STRING);

A string intrinsic 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).

FUNCTION COS (X: NUMERIC): REAL;

An arithmetic function. Returns the cosine of X in radians. Both X and the return value are of type REAL. To force a particular precision, declare CNSRQQ (CONSTSS REAL4) and/or CNDRQQ (CONSTS REAL8) and use them instead.

PROCEDURE DATE (VAR S: STRING);

A clock procedure. If available, this procedure assigns the current date to its STRING (or LSTRING) variable. If an LSTRING is passed as the parameter, you must set the length you want before calling the procedure. The format depends on the target operating system.

FUNCTION DECODE (CONST LSTR: LSTRING, X:M:N): BOOLEAN;

An extend level intrinsic 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.

DECODE works exactly the same as the READ procedure, including the use of M and N parameters (see Section 15.2.2 for a discussion of these parameters). 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).

In a segmented memory environment, the LSTR parameter must reside in the default data segment.

See also FUNCTION ENCODE.

PROCEDURE DELETE (VARS D: LSTRING; I, N: INTEGER);

A string intrinsic 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 DISBIN;

A library routine (interrupt). Along with ENABIN and VECTIN, DISBIN handles interrupt processing. DISBIN disables interrupts; ENABIN enables interrupts; VECTIN sets an interrupt vector. The effect of these procedures varies with the target machine. See Appendix A in the MS-Pascal User's Guide for information about your implementation.

PROCEDURE DISCARD (VAR F);

A file system procedure (extend level I/O). Closes and deletes an open file.

See Section 15.3.1 for a decription of DISCARD.

PROCEDURE DISPOSE (VARS P: POINTER);

A dynamic allocation procedure (short form). 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 check 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 can 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 MS-Pascal.

PROCEDURE DISPOSE
(VARS P: POINTER; T1, T2, ... TN: TAGS);

A dynamic allocation procedure (long form). The long form of DISPOSE works the same 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 ENABIN;

A library routine (interrupt handling). Along with DISBIN and VECTIN, ENABIN handles interrupt processing. ENABIN enables interrupts; DISBIN disables interrupts; VECTIN sets an interrupt vector. The effect of these procedures may vary with the target machine. See Appendix A in the MS-Pascal User's Guide for information about your implementation.

FUNCTION ENCODE (VAR LSTR: LSTRING, X:M:N): BOOLEAN;

An extend level intrinsic 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 Section 15.2.4 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).

In a segmented memory environment, the LSTR parameter must reside in the default data segment.

See also FUNCTION DECODE.

PROCEDURE ENDOQQ;

A library procedure (termination). ENDOQQ is called during termination, and the default version does nothing. However, you can 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.

See also PROCEDURE BEGOQQ.

PROCEDURE ENDXQQ;

The termination procedure. ENDXQQ is the overall termination routine and performs the following actions:

1. Calls ENDOQQ.

2. Terminates the file system (closing any open files).

3. Returns to the target operating system (or whatever called BEGXQQ).

ENDXQQ can be useful for ending program execution from inside a procedure or function, without calling ABORT. ENDXQQ corresponds to the HALT procedure in other Pascals.

FUNCTION EOF: BOOLEAN;
FUNCTION EOF (VAR F): BOOLEAN;

A file system function. Indicates whether the current position of the file is at the end of the file F for SEQUENTIAL and TERMINAL file modes. EOF with no parameters is the same as EOF (INPUT).

See Section 15.1.3 for a more complete description of EOF.

FUNCTION EOLN: BOOLEAN;
FUNCTION EOLN (VAR F): BOOLEAN;

A file system function. Indicates whether the current position of the file is at the end of a line in the textfile F. EOLN with no parameters is the same as EOLN (INPUT).

See Section 15.1.3 for a description of EOLN.

PROCEDURE EVAL (EXPRESSION, EXPRESSION, ...);

An extend level intrinsic 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 EXP (X: NUMERIC): REAL;

An arithmetic 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, declare EXSRQQ (CONSTS REAL4) and/or EXDRQQ (CONSTS REAL8) and use them instead.

PROCEDURE FILLC (D: ADRMEM; N: WORD; C: CHAR);

A system level intrinsic procedure. Fills D with N copies of the CHAR C. No bounds checking is done.

See also PROCEDURE FILLSC for segmented address types. 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);

A system level intrinsic procedure. Fills D with N copies of the CHAR C. No bounds checking is done.

See also PROCEDURE FILLC for relative address types. 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 FLOAT (X: INTEGER): REAL;

A data conversion 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 run-time package, it is included at the standard level.

FUNCTION FLOAT4 (X: INTEGER4): REAL;

A data conversion function. Converts an INTEGER4 value to a REAL value. This type conversion is also done automatically; however, it is possible to lose precision. (Losing precision is not an error.)

FUNCTION FREECT (SIZE: WORD): WORD;

A library function. Returns an estimate of the number of times NEW could be called to allocate heap variables with length SIZE bytes. FREECT takes into account DISPOSE and adjacent free blocks and is generally used with the SIZEOF function. However, it does not assume any stack space will be needed. Since stack space generally will be needed, the value returned should be reduced accordingly.

Example:

```
IF FREECT (SIZEOF (REC, TRUE, 5)) > 2
  THEN DO_SOMETHING
```

PROCEDURE GET (VAR F);

A file system procedure. GET either reads the currently pointed-to component of F to the buffer variable F^ and advances the file pointer, or sets the buffer variable status to empty.

See Section 15.1.1 for a description of GET.

FUNCTION GTYUQQ (LEN: WORD; LOC: ADSMEM): WORD;

A library function (terminal I/O). Reads a maximum of LEN characters from the terminal keyboard and stores them in memory beginning at the address LOC. The return value is the number of characters actually read. GTYUQQ always reads the entire line you enter. Any characters typed beyond the end of the buffer length are lost.

Example:

```
LSTR.LEN := GTYUQQ (UPPER(LSTR), ADS LSTR(1));
```

Together with PTYUQQ and PLYUQQ, GTYUQQ is useful for doing terminal I/O in a low-overhead environment. These functions are part of a collection of routines called Unit U, which implements the MS-Pascal file system. (See Section 8.2 in your MS-Pascal User's Guide for further information on Unit U.)

FUNCTION HIBYTE (INTEGER-WORD): BYTE;

An extend level intrinsics function. Returns the most significant byte of an INTEGER or WORD. Depending on the target processor, the most significant byte may be the first or the second addressed byte of the word.

See also FUNCTION LOBYTE.

FUNCTION HIWORD (INTEGER4): WORD;

An extend level intrinsics 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.

See also FUNCTION LOWORD.

PROCEDURE INSERT
(CONSTS S:STRING; VARS D:LSTRING; I:INTEGER);

A string intrinsic 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

FUNCTION LADDOK
(A, B: INTEGER4; VAR C: INTEGER4): BOOLEAN;

A library routine (no-overflow arithmetic). Sets C equal to A plus B. One of two functions that do 32-bit signed arithmetic without causing a run-time 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 LDSRQQ (CONSTS A: REAL4): REAL4;
FUNCTION LDDRQQ (CONSTS A: REAL8): REAL8;

Arithmetic functions. Return the logarithm, base 10, of A. Both A and the return value are of type REAL4 or REAL8, as shown. These functions are from the MS-FORTRAN run-time library and must be declared EXTERN before use.

FUNCTION LMULOK
(A, B: INTEGER4; VAR C: INTEGER4): BOOLEAN;

A library routine (no-overflow arithmetic). Sets C equal to A times B. One of two functions that do 32-bit signed arithmetic without causing a run-time error overflow. Normal arithmetic may cause a run-time 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 LN (X: REAL): REAL;

An arithmetic function. Returns the logarithm, base e, of X. Both X and the return value are of type REAL. To force a particular precision, 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 LOBYTE (INTEGER-WORD): BYTE;

An extend level intrinsic function. Returns the least significant byte of an INTEGER or WORD. Depending on the target processor, the least significant byte may be the first or the second addressed byte of the word.

See also FUNCTION HIBYTE.

FUNCTION LOCKED (VARS SEMAPHORE: WORD): BOOLEAN;

A library function (semaphore). If the semaphore is available, LOCKED returns the value TRUE and sets the semaphore unavailable. Otherwise, if it is

already locked, LOCKED returns FALSE. UNLOCK sets the semaphore available. As a binary semaphore, there are only two states.

See also PROCEDURE UNLOCK.

FUNCTION LOWER (EXPRESSION): VALUE;

An extend level intrinsic function. LOWER 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.

See also FUNCTION UPPER.

FUNCTION LOWORD (INTEGER4): WORD;

An extend level intrinsic function. Returns the low-order WORD of the four bytes of the INTEGER4.

See also FUNCTION HIWORD.

PROCEDURE MARKAS (VAR HEAPMARK: INTEGER4);

A library procedure (heap management). Parallels the MARK procedure in other Pascals. MARKAS marks the upper and lower limits of the heap. The DISPOSE

procedure is generally more powerful, but MARKAS may be useful for converting from other Pascal dialects.

In other Pascals, the parameter is of a pointer type. However, MS-Pascal needs two words to save the heap limits, since in some implementations the heap grows toward both higher and lower addresses. The HEAPMARK variable should not be used as a normal INTEGER4 number; it should only be set by MARKAS and passed to RELEAS.

To use MARKAS and RELEAS, pass an INTEGER4 variable, say M, as a VAR parameter to MARKAS. MARKAS places the bounds of the heap in M. To release heap space, simply invoke the procedure with RELEAS (M).

MARKAS and RELEAS work as intended only if you never call DISPOSE.

FUNCTION MDSRQQ (CONSTS A, B: REAL4): REAL4;
FUNCTION MDDRQQ (CONSTS A, B: REAL8): REAL8;

Arithmetic functions. 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, as shown. These functions are from the MS-FORTRAN run-time library and must be declared EXTERN before use.

FUNCTION MEMAVL: WORD;

A library function (heap management). Returns the number of bytes available between the stack and the heap. MEMAVL acts like the MEMAVAIL function in UCSD Pascal. If you have previously used DISPOSE, MEMAVL may return a value less than the actual number of bytes available.

FUNCTION MNSRQQ (CONSTS A, B: REAL4): REAL4;
FUNCTION MNDRQQ (CONSTS A, B: REAL8): REAL8;

Arithmetic functions. Return the value of A or B, whichever is smaller. Both A and B are of type REAL4 or REAL8, as shown. These functions are from the MS-FORTRAN run-time library and must be declared EXTERN before use.

See also FUNCTION MXSRQQ and FUNCTION MXDRQQ.

PROCEDURE MOVEL (S, D: ADRMEM; N: WORD);

A system level intrinsic 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)
```

See also PROCEDURE MOVESL for segmented address types. 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);
A system level intrinsic procedure. 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)
```

See also PROCEDURE MOVESR for segmented address types.

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);

A system level intrinsic 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:

```
MOVESL (ADS 'New String Value', ADS V, 16)
```

See also PROCEDURE MOVEL for relative address types. 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);

A system level intrinsic procedure. 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)
```

See also PROCEDURE MOVER for relative address types.

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 MXSRQQ (CONSTS A, B: REAL4): REAL4;
FUNCTION MXDRQQ (CONSTS A, B: REAL8): REAL8;

Arithmetic functions. Return the value of A or B, whichever is larger. Both A and B are of type REAL4 or REAL8, as shown. These functions are from the MS-FORTRAN run-time library and must be declared EXTERN before use.

See also FUNCTION MNSRQQ and MNDRQQ.

PROCEDURE NEW (VARS P: POINTER);

A library procedure (heap management, short form). 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, use the long form of the procedure instead. 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);

A library procedure (heap management, long form). 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, MS-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 run-time 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:

- Assign tag values
- Check that they are initialized correctly
- Check that their value is not changed during execution

According to the ISO standard, a variable created with the long form of NEW cannot be:

- Used as an expression operand
- Passed as a parameter
- Assigned a value

MS-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 wipes out part of the heap. This condition is difficult to detect at compile-time. Therefore, in MS-Pascal, 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, and the tag checking switch is on.

The extend 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.

FUNCTION ODD (X: ORDINAL): INTEGER;

A data conversion 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;

A data conversion function. Converts to INTEGER any value of of one of the types shown in Table 14-6, according to the rules given.

Table 14-6: Conversion to INTEGER

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 s ORD(LOWORD(INTEGER4))
Pointer	Integer value of pointer

```
PROCEDURE PACK
(CONSTS A: UNPACK-ARRAY; I: INDEX; VARS
          Z: PACKED-ARRAY);
```

A data conversion 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.

```
PROCEDURE PAGE;
PROCEDURE PAGE (VAR F);
```

A file system procedure. Causes skipping to the top of a new page when the textfile F is printed. PAGE with no parameter is the same as PAGE (INPUT).

See Section 15.1.4 for a description of PAGE.

```
FUNCTION PISRQQ (CONSTS A: REAL4; CONSTS
          B: INTEGER4): REAL4;
FUNCTION PIDRQQ (CONSTS A: REAL8; CONSTS
          B: INTEGER4): REAL8;
```

Arithmetic functions. The return value is A**B (A to the INTEGER power of B). A is of type REAL4 or REAL8, as shown. B is always of type INTEGER4. These functions are from the MS-FORTRAN run-time library and must be declared EXTERN before use.

PROCEDURE PLYUQQ;

A library routine (terminal I/O). Writes an end-of-line character to the terminal screen.

Together with GETYQQ and PTYUQQ, PLYUQQ is useful for doing terminal I/O in a low-overhead environment. These functions are part of a collection of routines called Unit U, which implements the MS-Pascal file system. (See Section 8.2 in your MS-Pascal User's Guide for further information on Unit U.)

FUNCTION POSITN
(CONSTS PAT: STRING; CONSTS
S: STRING; I: INTEGER): INTEGER;

A string intrinsic 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 PRED (X: ORDINAL): ORDINAL;

A data conversion 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 PRSRQQ (A, B: REAL4): REAL4;
FUNCTION PRDRQQ (A, B: REAL8): REAL8;

Arithmetic functions. The return value is A**B (A to the REAL power of B). Both A and B are of type REAL4 or REAL8, as shown. An error occurs if A < Ø (even if B happens to have an integer value). These functions are from the MS-FORTRAN run-time library and must be declared EXTERN before use.

PROCEDURE PTYUQQ (LEN: WORD; LOC: ADSMEM);

A library routine (terminal I/O). Writes LEN characters, beginning at LOC in memory, to the terminal screen.

Example:

```
PTYUQQ (8, ADS 'PROMPT: ');
```

Together with GETYQQ and PLYUQQ, PTYUQQ is useful for doing terminal I/O in a low-overhead environment. These functions are part of a collection of routines called Unit U, which implements the MS-Pascal file system. (See Section 8.2 in your MS-Pascal User's Guide for further information on Unit U.)

PROCEDURE PUT (VAR F);

A file system procedure. Writes the value of the file buffer variable F^ to the currently pointed-to component of F and advances the file pointer.

See Section 15.1.1 for a description of PUT.

PROCEDURE READ (F)

A file system procedure. READ reads data from files. Both READ and READLN are defined in terms of the more primitive operation, GET.

See Section 15.2 for a description of READ.

PROCEDURE READFN (VAR F; P1, P2, ... PN);

A file system procedure (extend level I/O). READFN 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).

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.

See Section 15.3.1 for a description of READFN.

PROCEDURE READLN (F)

A textfile I/O procedure. At the primitive GET level, without parameters, READLN (F) is equivalent to the following:

```
BEGIN
  WHILE NOT EOLN (F) DO GET (F);
  GET (F)
END
```

The procedure READLN is very much like READ, except that it reads up to and including the end of line.

See Section 15.2 for a description of READ.

PROCEDURE READSET
(VAR F; VAR L: LSTRING; CONST S: SETOFCHAR);

A file system procedure (extend level I/O). READSET reads characters and puts them into L, as long as the characters are in the set S and there is room in L.

See Section 15.3.1 for a description of READSET.

PROCEDURE RELEAS (VAR HEAPMARK: INTEGER4);

A library routine (heap management). Parallels the RELEASE procedure in other Pascals. RELEAS disposes of heap space past the area set with a previous MARKAS call. The DISPOSE procedure in MS-Pascal is generally more powerful, but RELEAS may be useful for converting from other Pascal dialects.

In other Pascals, the parameter is of a pointer type. However, MS-Pascal needs two words to save the heap limits, since in some implementations the heap grows toward both higher and lower addresses. The HEAPMARK variable should not be used as a normal INTEGER4 number; it should only be set by MARKAS and passed to RELEAS.

To use MARKAS and RELEAS, pass an INTEGER4 variable, say M, as a VAR parameter to MARKAS. MARKAS places the bounds of the heap in M. To RELEAS heap space, simply invoke the procedure with RELEAS (M).

MARKAS and RELEAS work as intended only if DISPOSE is never called.

PROCEDURE RESET (VAR F);

A file system procedure. Resets the current file position to its beginning and does a GET (F).

See Section 15.1.2 for a description of RESET.

FUNCTION RESULT (FUNCTION-IDENTIFIER): VALUE;

An extend level intrinsic function. Used to access the current value of a function; can be used only within the body of the function itself or in a procedure or function nested within it.

FUNCTION RETYPE (TYPE-IDENT, EXPRESSION): TYPE-IDENT;

A system level intrinsic 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 MS-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. Note that the relative mapping of variables is subject to change between different versions of the compiler.)

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.

PROCEDURE REWRITE (F);

A file system procedure. Resets the current file position to its beginning.

See Section 15.1.2 for a description of REWRITE.

FUNCTION ROUND (X: REAL): INTEGER;

An arithmetic function. Rounds X away from zero. X is of type REAL4 or REAL8; the return value is of type INTEGER. The effect of ROUND on a number with a fractional part of Ø.5 varies with the implementation.

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;

An arithmetic function. Rounds real X away from zero. X is of type REAL4 or REAL8; the return value is of type INTEGER4. The effect of ROUND4 on a number with a fractional part of Ø.5 varies with the implementation.

Examples:

ROUND4 (1.6) is 2

ROUND4 (-1.6) is -2

An error occurs if ABS (X + Ø.5) >=MAXINT4.

FUNCTION SADDOK
(A, B: INTEGER; VAR C: INTEGER): BOOLEAN;

A library routine (no-overflow arithmetic). Sets C equal to A plus B. One of two functions that do 16-bit signed arithmetic without causing a run-time error on overflow. Normal arithmetic may cause a

run-time 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 SCANEQ
(LEN: INTEGER; PAT: CHAR; CONSTS S: STRING;
I: INTEGER):INTEGER;

A string intrinsic 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; CONSTS S: STRING;
I: INTEGER):INTEGER;

A string intrinsic function. 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.

PROCEDURE SEEK (VAR F; N: INTEGER4);

A file system procedure (extend level I/O). In contrast to normal sequential files, DIRECT files are random access structures. SEEK is used to randomly access components of such files.

See Section 15.3 for details.

FUNCTION SHSRQQ (CONSTS A: REAL4): REAL4;
FUNCTION SHDRQQ (CONSTS A: REAL8): REAL8;

Arithmetic functions. Return the hyperbolic sine of A. A is of type REAL4 or REAL8, as shown. These functions are from the MS-FORTRAN run-time library and must be declared EXTERN before use.

FUNCTION SIN (X: NUMERIC): REAL;

An arithmetic function. Returns the sine of X in radians. Both X and the return value are of type REAL. To force a particular precision, declare SNSRQQ (CONSTS REAL4) and/or SNDRQQ (CONSTS REAL8) and use them instead.

FUNCTION SIZEOF (VARIABLE): WORD;
FUNCTION SIZEOF (VARIABLE, TAG1, TAG2, ...TAGN): WORD;

An extend level intrinsic 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 SMULOK
(A, B: INTEGER; VAR C: INTEGER): BOOLEAN;

A library routine (no-overflow arithmetic function). Sets C equal to A times B. One of two functions that do 16-bit signed arithmetic without causing a run-time error on overflow. Normal arithmetic may cause a run-time 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 SQR (X: NUMERIC): NUMERIC;

An arithmetic function. Returns the square of X, where X is of type REAL, INTEGER, WORD, or INTEGER4.

FUNCTION SQRT (X): REAL

An arithmetic function. Returns the square root of X, where X is of type REAL. To force a particular precision, declare SRSRQQ (CONSTS REAL4) and/or SRDRQQ (CONSTS REAL8) and use them instead. An error occurs if X is less than 0.

FUNCTION SUCC (X: ORDINAL): ORDINAL;

A data conversion 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 THSRQQ (CONSTS A: REAL4): REAL4;
FUNCTION THDRQQ (CONSTS A: REAL8): REAL8;

Arithmetic functions. Return the hyperbolic tangent of A. Both A and the return value are of type REAL4 or REAL8, as shown. These functions are from the MS-FORTRAN run-time library and must be declared EXTERN before use.

FUNCTION TICS: WORD;

A library routine (clock function). If available, TICS returns the value of an operating system timing location. The result is in a time interval, such as hundredths of a second, depending on the target operating system.

PROCEDURE TIME (VAR S: STRING);

A library routine (clock function). If available, this procedure assigns the current time to its STRING (or LSTRING) variable. If the parameter is an LSTRING, you must set the length before you call the TIME procedure. The format depends on the target operating system.

See also PROCEDURE DATE.

FUNCTION TNSRQQ (CONSTS A: REAL4): REAL4;
FUNCTION TNDRQQ (CONSTS A: REAL8): REAL8;

Arithmetic functions. Return the tangent of A. Both A and the return value are of type REAL4 or REAL8, as shown. These functions are from the MS-FORTRAN run-time library and must be declared EXTERN before use.

FUNCTION TRUNC (X: REAL): INTEGER;

An arithmetic 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;

An arithmetic 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.

FUNCTION UADDOK (A, B: WORD; VAR C: WORD): BOOLEAN;

A library routine (no-overflow arithmetic function). Sets C equal to A plus B. One of two functions that do 16-bit unsigned arithmetic without causing a run-time error on overflow. Normal arithmetic may cause a run-time 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;

A library routine (no-overflow arithmetic function). Sets C equal to A times B. One of two functions that do 16-bit unsigned arithmetic without causing a run-time error on overflow. Normal arithmetic may cause a run-time 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.

PROCEDURE UNLOCK (VARS SEMAPHORE: WORD);

A library routine (semaphore procedure). UNLOCK sets the semaphore available. As a binary semaphore, there are only two states. UNLOCK can be called any number of times and can be used to initialize the semaphore.

See also FUNCTION LOCKED.

PROCEDURE UNPACK
(CONSTS Z: PACKED-ARRAY; VARS A: UNPACK-ARRAY;
I: INDEX);

A data conversion 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 UPPER (EXPRESSION): VALUE;

An extend level intrinsic function. UPPER, like LOWER, 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.

See also PROCEDURE LOWER.

PROCEDURE VECTIN (V: WORD; PROCEDURE I [INTERRUPT]);

A library routine (interrupt handling procedure). One of three procedures for processing interrupts. VECTIN sets an interrupt vector, so that interrupts of type V are connected to procedure I. (ENABIN enables interrupts and DISBIN disables interrupts.)

The effect of these procedures and the meaning of V varies with the target machine. See Appendix A, in the MS-Pascal User's Guide for information regarding your implementation.

FUNCTION WRD (X: VALUE): WORD;

A data conversion procedure. Converts to WORD any of the types shown in Table 14-7, according to the rules given.

Table 14-7: Conversion to WORD

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

PROCEDURE WRITE (F)
PROCEDURE WRITELN (F)

File system level intrinsic procedures. WRITEs data to files. WRITE and WRITELN are defined in terms of the more primitive operation PUT. WRITELN is the same as WRITE, except it also writes an end-of-line.

See Section 15.2.3 for descriptions of these procedures.

15. FILE-ORIENTED PROCEDURES AND FUNCTIONS

This chapter discusses all of the file I/O procedures and functions, as well as lazy evaluation and concurrent I/O, two special MS-Pascal features that facilitate your use of files.

The MS-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 shown in Table 15-1.

Table 15-1: File System Procedures and Functions

CATEGORY	PROCEDURES	FUNCTIONS
Primitive	GET	EOF
	PAGE	EOLN
	PUT	
	RESET	
	REWRITE	
Textfile I/O	READ	
	READLN	
	WRITE	
	WRITELN	
Extend Level I/O	ASSIGN	
	CLOSE	
	DISCARD	
	READSET	
	READFN	
	SEEK	

15.1 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. Two related topics are also discussed in this section: lazy evaluation and concurrent I/O.

In all descriptions below, F is a file parameter (files are always reference parameters), and F^ is the buffer variable.

In a segmented environment, all file variables operated on by these procedures must reside in the default data segment. This restriction increases the efficiency of file system calls.

GET and PUT: The primitive procedures GET and PUT read to and write from the buffer variable, F^. A 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.

RESET and REWRITE: The procedures RESET and REWRITE set and REWRITE the current position of a file to its beginning. RESET prepares for later GET and READ procedures. REWRITE prepares for later PUT and WRITE procedures.

EOF and ELON: The functions EOF and EOLN are used and EOLN to check for the end-of-file and end-of-line conditions. They EOLN return a BOOLEAN result. In general, these values indicate when to stop reading a line or a file.

PAGE: The procedure PAGE helps in formatting textfiles. It is not a necessary procedure in the same sense as GET and PUT.

15.1.1 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);

A primitive file system intrinsic procedure. 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 run-time 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);

A primitive file system intrinsic 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 immediately after a RESET or GET.

Exceptions to these rules cause errors to be generated. The value of F^ always becomes undefined after a PUT.

In MS-Pascal, the value of F^ after a PUT (F) may vary, depending on the target operating system and type of file. 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.

15.1.2 RESET AND REWRITE

The procedures RESET and REWRITE set the current position of a file to its beginning. RESET prepares for later GET and READ operations. REWRITE prepares for later PUT and WRITE operations.

PROCEDURE RESET (VAR F);

A primitive file system intrinsic 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.

In MS-Pascal, a RESET closes the file and then opens it in a way that depends 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);

A primitive file system intrinsic 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).

In MS-Pascal, 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. If possible, an existing file with the same name is not affected when a REWRITE error occurs, but with some target operating systems the existing file may be deleted. 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.

15.1.3 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;

A primitive file system intrinsic 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 in some operating systems where a particular terminal character generates an end-of-file status,

or if INPUT is reassigned to another file. Calling the EOF (F) function accesses the buffer variable F^.

FUNCTION EOLN: BOOLEAN;
FUNCTION EOLN (VAR F): BOOLEAN;

A primitive file system intrinsic 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 MS-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^.

15.1.4 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);

A primitive file system intrinsic 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 formfeed, CHR (12). If F has mode TERMINAL, the effect is defined by the target operating system interface, which usually also writes a form feed.

15.1.5 LAZY EVALUATION

Lazy evaluation is designed to solve a recurring problem in Pascal, specifically, how to READ 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 in MS-Pascal 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, you must first type a character at the keyboard to accommodate the GET procedure. Only then are you 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. Table 15-2 summarizes these rules.

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 screen 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.

15.1.6 CONCURRENT I/O

On operating systems that support it, concurrent I/O permits a GET or PUT procedure to initiate the I/O and immediately return to the calling program. It is used only for BINARY structure files.

The program can do computation while the buffer variable is being filled or emptied. The buffer variable has another special status value that can be "ready" or "busy." If the status is busy when the buffer variable needs to be accessed, the program must wait until the status becomes ready.

For example, the following program fragment reads the file IN_FILE, does some computation with the current value, and then writes it to the file OUT_FILE:

```
WHILE NOT EOF (IN_FILE) DO

  {Check for end of input and}
  {wait until IN_FILE^ ready.}

BEGIN
  READ (IN_FILE, BUFF);
  {IN_FILE^ is ready, so assign it to BUFF;}
  {start reading next component.}

OPERATE (BUFF);
  {Go process value during READ and WRITE.}

WRITE (OUT_FILE, BUFF);
  {Wait until OUT_FILE^ is ready,}
  {then assign BUFF and start writing.}

END
```

The preceding example uses READ and WRITE procedures. Note that the following two lines are equivalent:

```
READ (IN_FILE, BUFF)
BUFF := IN_FILE^;  GET (IN_FILE)
```

These two are also identical:

```
WRITE (OUT_FILE, BUFF)
OUT_FILE^ := BUFF; PUT (OUT_FILE)
```

Concurrent I/O applies to the procedures GET and PUT, as well as to the procedures READ and WRITE. In practice, it is unusual for the MS-Pascal run-time system to handle concurrency. See Appendix A in the MS-Pascal User's Guide for information regarding your implementation.

When accessing the buffer variable, either for lazy evaluation or concurrency, MS-Pascal generates an I/O system call. However, if the buffer variable is an actual reference parameter, the procedure or function using that parameter can do I/O to the same file, and these special calls cannot be executed.

Passing any buffer variable as a reference parameter is an error in MS-Pascal, although only a warning is given. Calling GET or PUT has an undefined effect on a file buffer variable accessed indirectly through a reference parameter. Assigning the address of a buffer variable to an address type variable is equally dangerous, since this bypasses the lazy evaluation and concurrency mechanisms.

15.2 TEXTFILE 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 extend 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.

READ and READLN: The procedures READ and READLN read data from textfiles. READ and READLN are defined in terms of the more primitive operation, GET. The procedure READLN is very much like READ, except that it reads up to and including the end of line.

WRITE and WRITELN: The procedures WRITE and WRITELN write data to textfiles. WRITE and WRITELN are defined in terms of the more primitive operation, PUT. The procedure WRITELN writes a line marker to the end of a line. In all other respects, WRITELN is analogous to WRITE.

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 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;
```

In MS-Pascal, 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.

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, be generated only by the procedure WRITELN. However, in MS-Pascal, an actual character may be used for the line marker, and it can therefore be possible to write one, 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 non-empty file in which the last character is not already a line marker.

When a textfile being read reaches the end of a non-empty 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 MS-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).

15.2.1 READ AND READLN

PROCEDURE READ
PROCEDURE READLN

File system intrinsic procedures for textfile I/O. READ and READLN read data from text files. 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,

```
BEGIN
  READ (F, P1, P2, Pn);
  READLN (F)
END
```

READLN is often used to skip to the beginning of the next line. It can be used only 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)
{Reads INPUT by default}
READ (INPUT, P1, P2, P3)
```

At the standard level, parameters P1, P2, and P3 above must be of one of the following types:

CHAR
INTEGER
REAL

The extend 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
```

15.2.2 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.

Three important points apply to formatted reads.

1. First, leading spaces, tabs, formfeeds, 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. Second, 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.

3. Third, M and N values in READ are ignored, except as noted for an N value with enumerated types. M and N parameters are not accepted in BINARY 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 forms 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 extend 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 INTEGER4s. 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 extend 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' causes true and false, respectively, to be assigned to P. The number read must be in the range of the ORD values of the variable.

Also at the extend level, if the parameter P is an enumerated type and includes the :N notation as in READ (P::N), characters are read from the file F that form a valid identifier or number. If the characters form a number, it is assumed to be the ORD value (above), and if the characters form an identifier that is one of the enumerated type's constant identifiers, its value is assigned to P. In addition, if the variable is BOOLEAN, reading one of the digits 1 or Ø causes either true or false to be assigned to the BOOLEAN variable. 'TRUE' and

'FALSE' are also accepted as the BOOLEAN constant identifiers.

The actual value of N is ignored: using the N notation directs the compiler to save the enumerated type's constant identifiers and make them available to the applicable READ routine. Omitting the N notation saves memory that would be used for the identifiers.

4. Reference types

At the extend level, if P is a pointer type, a number is read as a WORD and assigned to P, in an implementation defined way such 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 extend level, if P is a STRING (n), then the next "n" characters are read sequentially into P. Preceding line 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 extend 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 formfeeds are not skipped. If the line marker is encountered before n characters are 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.

Currently, READ and READLN do not use M field width parameters: you cannot read the line '123456' as two INTEGER numbers with READ (I:3, J:3). However, you can read two LSTRING (3) items and then decode them to achieve the same effect.

15.2.3 WRITE AND WRITELN

PROCEDURE WRITE
PROCEDURE WRITELN

File system intrinsic procedures (textfile I/O). 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,

```
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
INTEGER
REAL
BOOLEAN
STRING

At the extend level, expressions can also be of the following types:

WORD
INTEGER4
LSTRING
an enumerated type
a pointer type

Parameters can take optional M and N values (see Section 15.2.4 for information about M and N parameters).

The procedure WRITE can also write to a BINARY file (i.e., not a textfile). For DIRECT files after a SEEK operation, however, 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. Normally, BINARY writes do not accept M and N values.

15.2.4 WRITE FORMATS

In textfiles, data parameters to WRITE and WRITELN can 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 extend level permits M and N values for both READs and 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). Currently, 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 extend 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 MS-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 a hexadecimal, decimal, octal, binary, or other a 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 examples illustrate WRITE operations on REAL values:

This statement:	Produces this 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 extend 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 extend 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 extend 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 extend 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 fill the remaining positions, not characters past the length in the LSTRING. Note that a string of M blanks can be written with NULL:M.

15.3 EXTEND LEVEL I/O

At the extend level, MS-Pascal has these additional I/O features:

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.

Section 7.6 discusses FCB fields in the context of files. The additional procedures and temporary files are described in the following sections.

15.3.1 EXTEND LEVEL PROCEDURES

PROCEDURE ASSIGN (VAR F; CONSTS N: STRING);

A file system procedure (extend level I/O). Assigns an operating system filename in a STRING (or LSTRING) to a file F. The filename format depends on the target operating system. 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);

A file system procedure (extend level I/O). 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 MS-Pascal 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 run-time errors may remove control from the MS-Pascal run-time 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);

A file system procedure (extend level I/O). Closes and deletes an open file. DISCARD is much like CLOSE except that the file is deleted.

PROCEDURE READFN (VAR F: P1, P2, ... PN);

A file system procedure (extend level I/O). READFN 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);

A file system procedure (extend level I/O). READSET 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 run-time 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.

In a segmented memory environment, the L and S parameters must reside in the default data segment.

PROCEDURE SEEK (VAR F; N: INTEGER4);

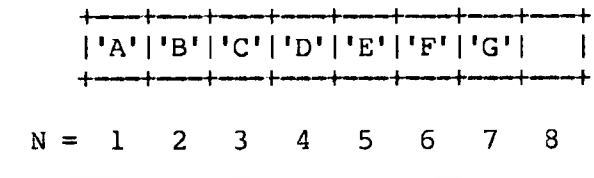
A file system procedure (extend level I/O). In contrast to normal sequential files, DIRECT files are random access structures. SEEK is used to randomly access components of such files. To use a DIRECT file, the MODE field must be set to DIRECT before the file is opened with RESET or REWRITE; the file, 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:

Figure 15-1:



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'.}
```

Always followa 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 either ASCII or BINARY structured files. However, READ and WRITE work only with ASCII files,

i.e., textfiles. READ, in particular, does not work with DIRECT mode BINARY files, because it assigns the buffer variable's value before it performs a GET. Care should always be taken when mixing normal sequential operations with DIRECT mode SEEK operations.

15.3.2 TEMPORARY FILES

Sometimes a program needs a "scratch" file for temporary, intermediate data. If this is the case, you can 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.

16. COMPILABLE PARTS OF A PROGRAM

MS-Pascal Compiler can compile three kinds of source files: programs, modules, and implementations of units. Modules and implementations of units can be compiled separately and later be linked to a program without recompilation. At the standard level, you can compile only entire programs; modules and units are MS-Pascal features available at the extend level.

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 declared 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 declared 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 required at the start of any}
{source that implements or uses a unit.}

INTERFACE;
  UNIT UNIT_DEMO (UNIT_PROC);
  PROCEDURE UNIT_PROC;'
END;
PROGRAM MAIN (INPUT, OUTPUT);
{USES clause below needed to connect}
{implementation and program.}
USES UNIT_DEMO;

{EXTERN declaration needed to connect}
{module's procedure.}
PROCEDURE MOD_PROC; EXTERN;
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 pieces:

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 Sections 16.1, 16.2, and 16.3.

16.1 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 (10);

{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 extend level, you can 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, 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 INPUT and OUTPUT in the program but omit them as program parameters. Their only effect as program parameters is to suppress this warning.

You can 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 can 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 puts those characters effectively at the start of the file INPUT. The identifier of the parameter is passed to both routines (PPMFQQ and PPMUQQ). Some operating systems then use the identifier as a prompt.

The use of program parameters in MS-Pascal can be illustrated by showing how to change a program into a procedure. Suppose you have a program like the following:

```
PROGRAM ALPHA (INPUT, OUTPUT, P1, P2, ..., Pn);
<declarations>
{Including those for P1, P2, ..., Pn}
BEGIN
  <body>
END.
```

PROGRAM ALPHA could then become the following procedure:

```
PROCEDURE ENTGQQ [PUBLIC];

<declarations>
{Including those for P1, P2, ..., Pn}
BEGIN
  PPMFQQ ('P1');  READFN (INPUT, P1);
  PPMFQQ ('P2');  READFN (INPUT, P2);
  .
  .
  PPMFQQ ('Pn');  READFN (INPUT, Pn);
  PPMEQQ ;
  {Called after all parameters are read}
  <program statements>
END;
```

The action of the interface routine PPMFQQ depends on the target operating system. See your MS-Pascal User's Guide for more information on PPMFQQ and ENTGQQ.

Some operating systems have elaborate mechanisms to handle this kind of parameter, using menus and default values. If yours falls into this category, the same mechanism generally applies to MS-Pascal program parameters.

Other less sophisticated operating systems pass to a program the remainder of the command line that invoked it; in this case, parameter values are read from the command line.

If the operating system does not provide a program parameter mechanism, or if an error occurs while using such a mechanism, or if it does not supply enough parameter values, then the PPMFQQ routine reverts to handling parameter values itself. It prompts you for every parameter with the parameter's identifier and reads the value you give it for the parameter. See Appendix A in your MS-Pascal User's Guide for details on how your implementation initializes program parameters.

16.2 MODULES

Modules provide a simple, straightforward method for combining several compilable segments into one program. Units, described in Section 16.3, 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 can 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 can specify them for individual procedures and functions.

4. If PURE or INTERRUPT are used, the module must contain only functions for PURE and procedures for INTERRUPT.

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 can 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 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.

16.3 UNITS

MS-Pascal units provide a structured way to access separately compiled modules. A unit has two parts:

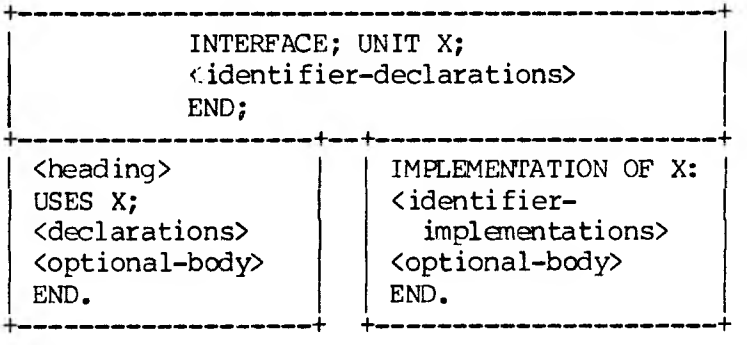
1. An interface

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. The general scheme is shown in Figure 16-1.

Figure 16-1: An MS-Pascal 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. In the above diagram, the INTERFACE is shared; the same INTERFACE exists in both the IMPLEMENTATION source file and in the other source file. Conversely, any other program, module, or unit could USE UNIT X; similarly, there could be

another IMPLEMENTATION OF X, in assembly language, for example.

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 MS-Pascal or one in assembly language). A large MS-Pascal program is often better organized as a main program and a number of units (parts of the MS-Pascal run-time system are organized in this way). However, only a program, module, interface, or implementation can USE a unit, not an individual procedure or function.

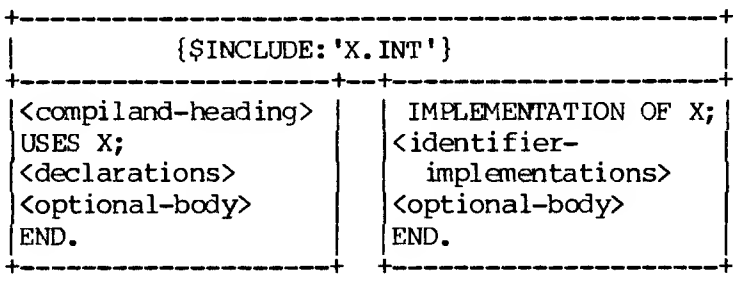
A program, module, implementation, or interface that uses an interface must start with the source file for that interface. Generally, the interface source file is a separate file, and an $INCLUDE metacommand at the start of the source file brings in the interface source itself at compile time. Because there is then only one master copy of the interface, this procedure 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 want several versions of the same interface. For example, there is a separate version of the MS-Pascal 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.

Suppose the INTERFACE for UNIT X in Figure 16.1 is contained in the file X.INT. If that is so, the compiland using the unit and the IMPLEMENTATION of the unit need only to $INCLUDE the interface file at

the start of the source file, as shown in Figure 16-2.

Figure 16-2: Unit with File X.INT



An MS-Pascal source file of any kind contains zero or more unit interfaces, separated by semicolons and followed by a program, a module, or an implementation, followed by a period. Each of these entities is called a "division." See Sections 16.3.1 and 16.3.2 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 with 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, can 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 can 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.

The following example shows 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 (0, 0);
    PLOT (10, 0);  PLOT (10, 10);
    PLOT (0, 10);  PLOT (0, 0);
  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, 0, 0)
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 can 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 associated with variables, procedures, and functions are associated with the corresponding identifiers in the interface. These become external references for the linker.

If the sample program above is compiled, every reference to the procedure PLOT generates an external reference to WJUMP. However, references to DRAWLINE 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.

16.3.1 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 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 pointless since NEWKEY is unknown even in 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. When you declare a file in an interface, the compiler does not give the usual warning:

Initialize Variable

If your interface contains files, be sure to end it with BEGIN END so that it will be initialized.

16.3.2 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 can 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 can 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 can implement a unit in assembly language, in which case all variables, procedures, and functions should generate public definitions for the loader. You can also implement units in other programming languages, such as MS-FORTRAN, or in a mixture of languages. If the interface is not implemented in MS-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 MS-Pascal run-time units are implemented partially in MS-Pascal and partially in assembly language. As mentioned, any IMPLEMENTATION section that does not implement all interface procedures and functions must declare those not implemented with the EXTERN directive at the start of the implementation.

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 clauses is 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:

- User initialization code
- A guarantee of only one initialization
- 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.

17. MS-PASCAL METACOMMANDS

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

- MS-Pascal language level
- Debugging and error handling
- Optimization level
- Use of the source file during compilation
- Listing file format

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

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

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

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

You can 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 MS-Pascal programs for use with other compilers, remember 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 $EXTEND.

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'.

Table 17-1 illustrates the notational conventions observed in the metacommand descriptions that follow.

Table 17-1: Metacommand Notation

NOTATION	MEANING
	Metacommand is typeless.
+ or -	Metacommand is an on/off switch. + sets value to 1 (on). - sets value to 0 (off). Default is indicated by + or - in heading.
:<n>	Metacommand is an integer.
:<'text'>	Metacommand is a string.

String values in the metalanguage can 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.

A complete alphabetical listing of MS-Pascal metacommands is given in Appendix G.

17.1 LANGUAGE LEVEL SETTING AND OPTIMIZATION

The metacommands shown in Table 17-2 let you control the level (standard, extend, or system) at which the compiler processes your program and the degree to which optimization is used. Some of these metacommands may not be implemented in your version of the compiler. See Appendix A in your MS-Pascal User's Guide for details.

Table 17-2: Language and Optimization Level

NAME	DESCRIPTION
$EXTEND	Adds extend level features.
$INTEGER:<n>	Sets the length of the INTEGER type.
$REAL:<n>	Sets the length of the REAL type.
$ROM	Gives a warning on static initialization.
$SIMPLE	Disables global optimizations.

$SIZE	Minimizes size of code generated.
$SPEED	Minimizes execution time of code.
$STANDARD	Enables standard level only.
$SYSTEM	Adds extend and system level features.

The compiler issues a warning message if it encounters a feature whose level is not enabled. The default setting is $EXTEND, which permits structured extensions that are relatively safe and portable. The default also requires you to explicitly request $SYSTEM extensions, which are by their nature low level, machine dependent, and relatively unstructured.

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

The effect of the $SIZE and $SPEED metacommands varies with the version of the optimizer in your implementation of the compiler. The default is $SIZE. If you select $SIMPLE, no optimization of any kind is done. $SIZE, $SPEED, and $SIMPLE are all mutually exclusive. 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 in Section 17.2)

17.2 DEBUGGING AND ERROR HANDLING

The metacommands shown in Table 17-3 are for debugging and error handling. They also generate code to check for run-time errors. Each of these metacommands is discussed in more detail on the following pages. Most of the metacommands in this group may also be given as command line switches to the compiler. See Section 17.7 for details.

Table 17-3: Debugging and Error Handling

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 run-time errors.
$STACKCK+	Checks for stack overflow at procedure or function entry.
$TAGCK-	Checks tag fields in variant records.
$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 run-time error invokes a call to the run-time 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 optional error status value, such as an operating system return code.

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

$DEBUG-

Turns on or off all of the debug switches (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. You can also 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. To 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:

- Uninitialized INTEGERs and 2-byte INTEGER subranges with the hexadecimal value 16#8000

- Uninitialized 1-byte INTEGER subranges with the hexadecimal value 16#80

- Uninitialized pointers with the value 1 (if $NILCK is also on)

- 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 meta-command 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). See Section 14.2 for descriptions of these functions.

$NILCK+

Checks for the following conditions:

- Dereferenced pointers whose values are NIL
- Uninitialized pointers if $INITCK is also on
- Pointers that are out of range
- 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:

- Assignment to subrange variables
- CASE statements without an OTHERWISE clause
- Actual parameters for the CHR, SUCC, and PRED functions
- Indices in PACK and UNPACK procedures
- Set and LSTRING assignments and value parameters
- 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 can link in a custom version of EMSEQQ, the error message routine, to do something different (such as invoke the run-time debugger or reset a controller). For more information on error handling, see Chapter 8 in your MS-Pascal User's Guide.

$STACKCK+

Checks for stack overflow when entering a procedure or function and when pushing parameters larger than four bytes on the stack. In some implementations, stack overflow is always checked. In some implementations, stack overflow is never checked in procedures with the INTERRUPT attribute.

$TAGCK-

Checks tag values when accessing a variant field. Only those tag fields with identifiers (whose value is actually stored in the record) are checked.

$WARN+

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

17.3 SOURCE FILE CONTROL

A small group of 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 after the table.

Table 17-4: Source File Control

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

$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 MSDOS $THEN}
SECTOR = S12;
{$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), and so on. 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, target processor, and so on. Compilation can 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;
```

You can also respond with a quoted string literal to create a constant of type STRING (n). For example, the source file metacommand $INCONST:HEADER prompts for a header. By enclosing a literal string constant in quotes, you declare a string constant:

```
Inconst:  HEADER = 'Processor Version 2.75'
```

$MESSAGE

Allows you to send messages to your screen 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 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 files for saving and restoring the metacommands in the main source file.

17.4 LISTING FILE CONTROL

The metacommands listed in Table 17-5 and described in this section allow you to format the listing file as you wish.

Table 17-5: Listing File Control Metacommands

METACOMMAND	DESCRIPTION
$LINESIZE:<n>	Sets width of listing. Default is 79 or 131, depending on implementation.
$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 either 131 or 79, depending on the implementation. See Appendix A in your MS-Pascal User's Guide for the default on your system.

$LIST+

Turns on the source listing. Except for $LIST-, metacommands themselves appear in the listing. The format of the listing file is described in Section 17.5.

$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 Section 17.5). 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 framepointer 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 in Table 17-6.

Table 17-6: Symbol Table Notation

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.

17.5 LISTING FILE FORMAT

The following discussion of listing file format is keyed to this sample listing:

```
User Title                                                    PAGE   1
User Subtitle                                                12/11/82
                                                             10:49:17
JG IC   Line#   Source Line       MS-Pascal Version 3.0  10/82
   00       1   PROGRAM foo; {$symtab+}
   10       2   VAR i: integer; k: ARRAY [-9..0] OF integer,
             2   --------------------Warning 156 , Assumed ;^
   20       3   FUNCTION bar (VAR j: integer): integer;
   20       4     VAR k: ARRAY [0..9] OF integer;
   20       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      10       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}
   10      14     END;
           14   ----^306 Function Assignment Not Found

Symtab     14   Offset Length   Variable - BAR
                -     2      24   Return offset, Frame length
                -     2       2   (function return):Integer
                +     4       2   J                 :Integer VarP
                -    22      20   K                 :Array

   10      15   BEGIN
   11      16     i := bar (i);
   00      17   END.

Symtab     17   Offset Length  Variable
                    0      24   Return offset, Frame length
                    2       2   I                 :Integer
                    4      20   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 at the right side of the first line of the heading. In some versions, the date and time appear at the right side of the second and third line, respectively. 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 run-time errors.

Two kinds of compiler messages appear in the listing: errors and warnings. A compilation with any errors cannot generate code. A compilation with only warnings 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 H 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 messages than fit on the screen, the first ones 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 it cannot recover from, 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. (See Appendix A in your MS-Pascal User's Guide for the nesting level limit for your implementation.)

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.

17.6 COMMAND LINE SWITCHES

Many of the debugging and error handling metacommands described in Section 17.1 can also be given as switches at compile-time. You can give the switches either on the compiler command line or in response to prompts anywhere that spaces can go. Table 17-7 lists the metacommands available as compiler switches. See your MS-Pascal User's Guide for more information on using switches.

Table 17-7: Command Line Switches

SWITCH	METACOMMAND	DESCRIPTION
/A	$INDEXCK	Checks for array index values in range (including super array indices).
/D	$DEBUG	Turns on all other switches, including $ENTRY and $LINE.
/E	$ENTR	Generates procedure entry and exit calls for the debugger.
/L	$LINE	Generates line number calls for error checking.
/I	$INITCK	Checks for use of uninitialized values.
/M	$MATHCK	Checks for mathematical errors, such as overflow and division by zero.
/N	$NILCK	Checks for invalid pointer values, including NIL.
/Q	$DEBUG	Turns off all other switches, including $ENTRY and $LINE.
/R	$RANGECK	Checks for subrange validity, including assignments.
/S	$STACKCK	Checks for stack overflow at procedure or function entry.
/T	$TAGCK	Checks tagfields in variant records.

You can use the /Q switch, in combination with the others, to tailor your compilation to your needs. First, turn off all of the other switches and then selectively turn on only the ones you want or need.

APPENDIX A

PASCAL SYNTAX DIAGRAMS

The diagrams on the following pages show the fundamental syntax of the MS-Pascal language. They are arranged in the order that you would be likely to use the elements while writing a program. The meaning of the differently shaped outlines is as follows:

1. Ovals

   Indicate reserved words or symbols of the MS-Pascal language. These must be typed as shown.

2. Boxes

   Indicate higher-level constructions that usually have syntax diagrams of their own.

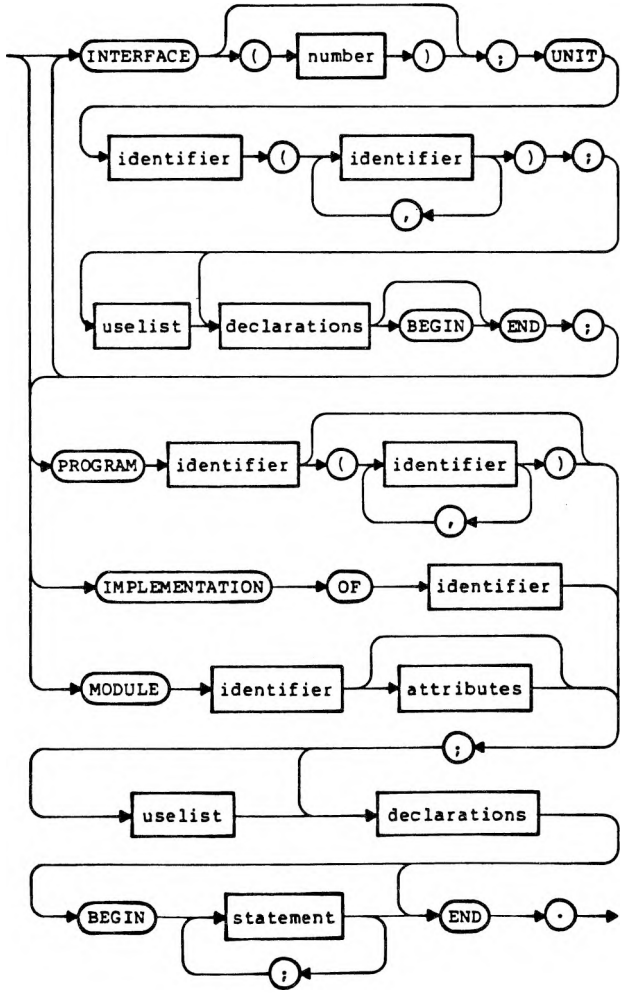
3. Circles

   Indicate punctuation that is required and must be typed as shown.

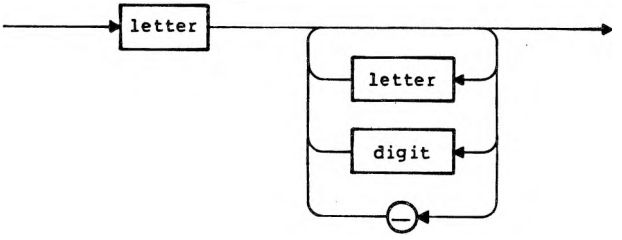
4. Arrows

   Help to show the path through the diagram, including any possible looping (i.e., repetition of syntax elements).

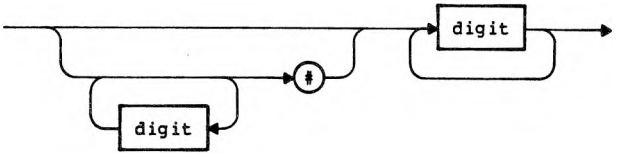
Source File


INTERFACE
(
number
)
;
UNIT
identifier
(
identifier
)
;
,
uselist
declarations
BEGIN
END
;
PROGRAM
identifier
(
identifier
)
,
IMPLEMENTATION
OF
identifier
MODULE
identifier
attributes
;
uselist
declarations
BEGIN
statement
END
.
;

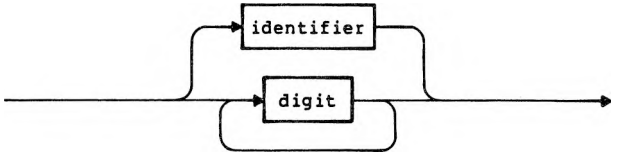
Identifier



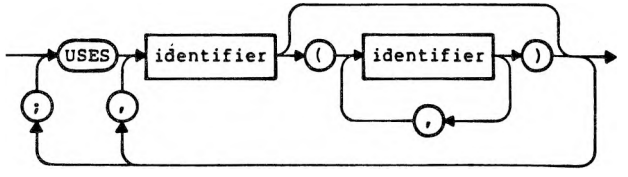
Number



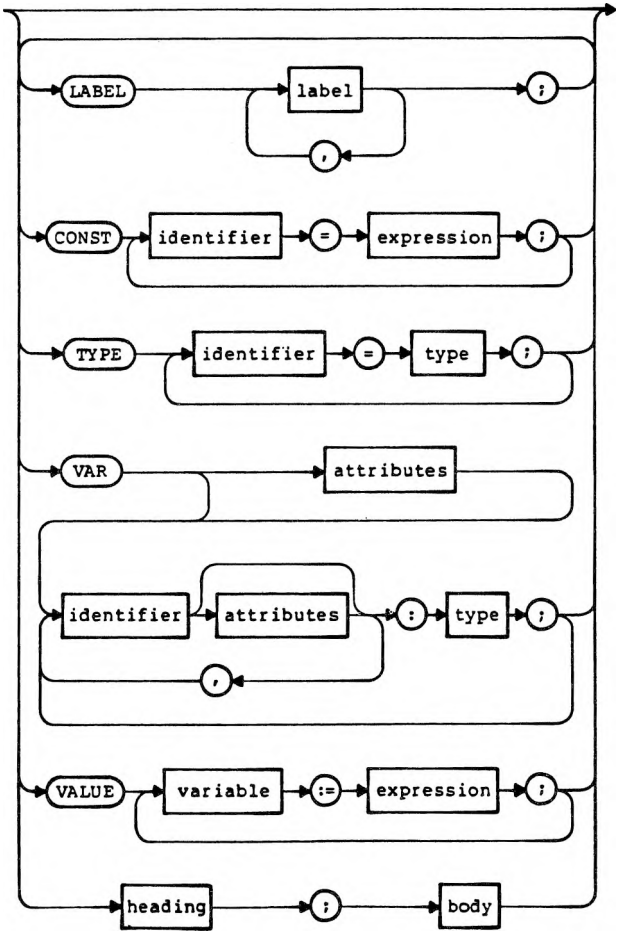
Label



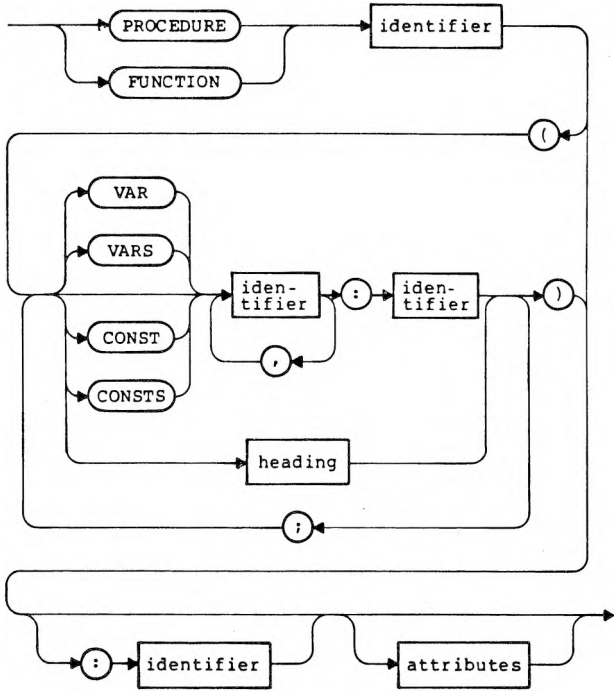
Uselist



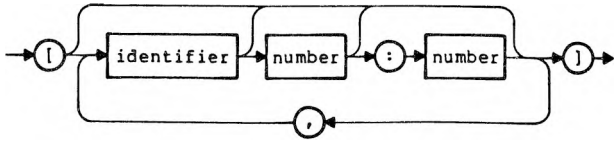
Declarations



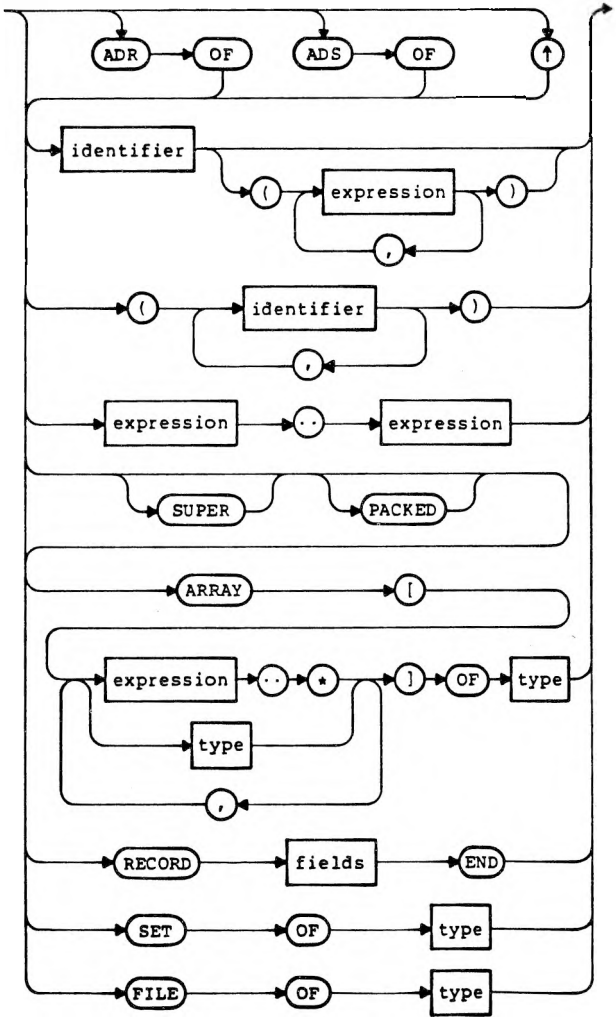
Heading



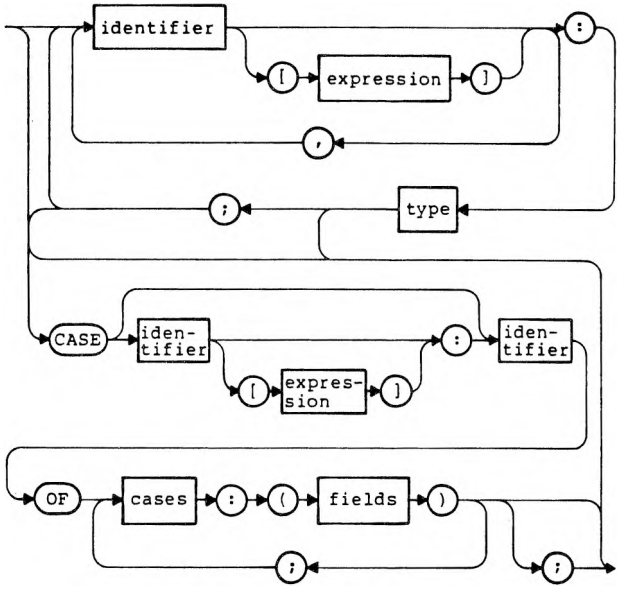
Attributes



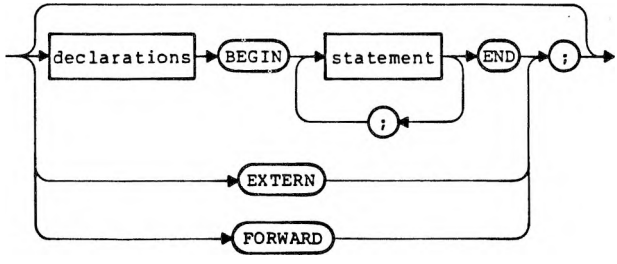
Type


ADR
OF
ADS
OF
↑
identifier
(
expression
)
,
(
identifier
)
,
expression
..
expression
SUPER
PACKED
ARRAY
[
expression
..
*
]
OF
type
type
,
RECORD
fields
END
SET
OF
type
FILE
OF
type

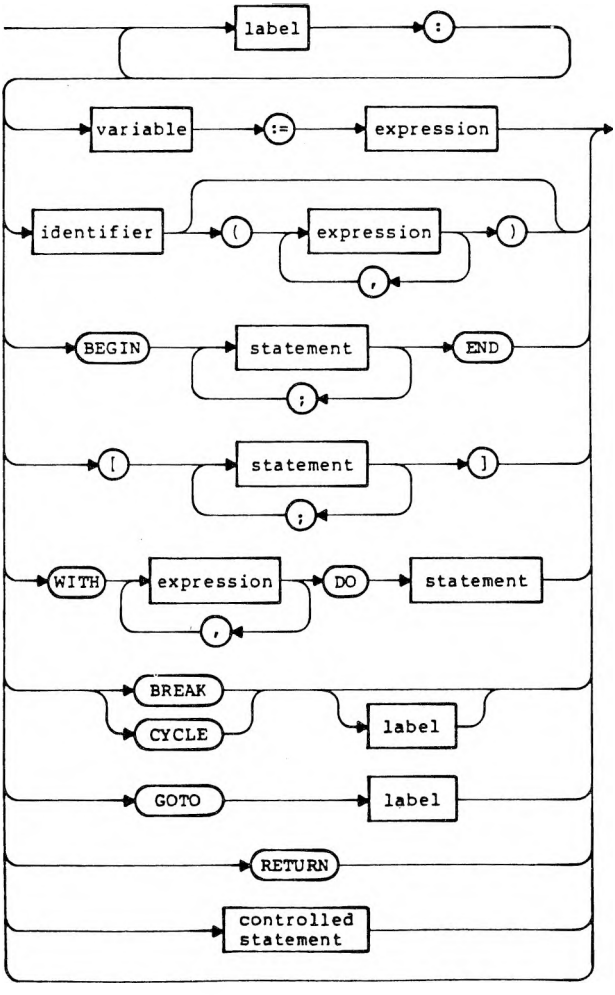
Fields


identifier
[
expression
]
:
,
;
type
CASE
iden-
tifier
[
expres-
sion
]
:
iden-
tifier
OF
cases
:
(
fields
)
;
;

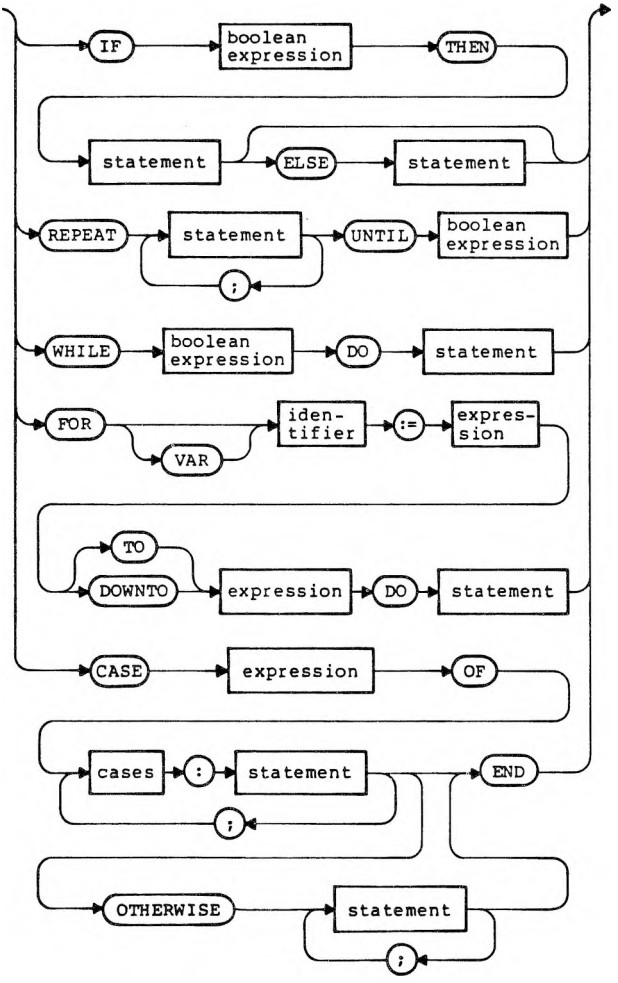
Body


declarations
BEGIN
statement
END
;
;
EXTERN
FORWARD

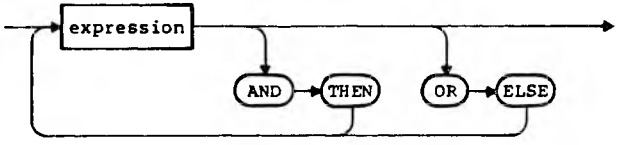
Statement


label
:
variable
:=
expression
identifier
(
expression
)
,
BEGIN
statement
END
;
[
statement
]
;
WITH
expression
DO
statement
,
BREAK
CYCLE
label
GOTO
label
RETURN
controlled
statement

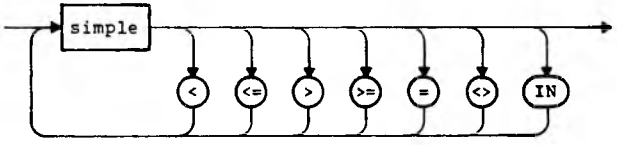
Controlled Statement



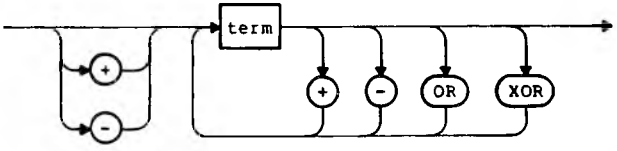
Boolean Expression



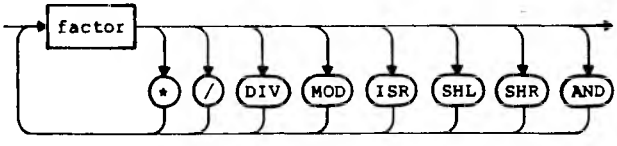
Expression



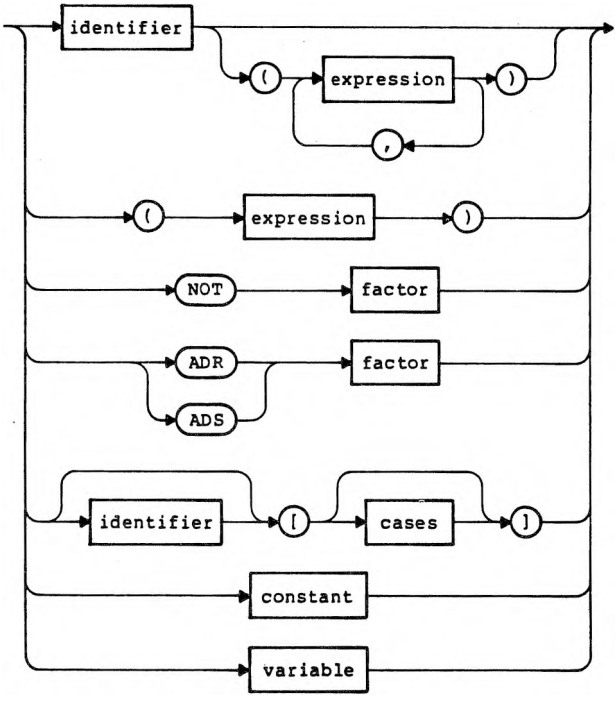
Simple



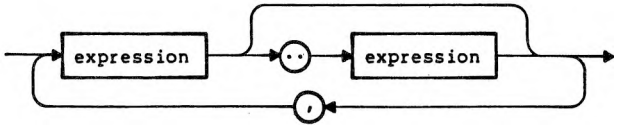
Term



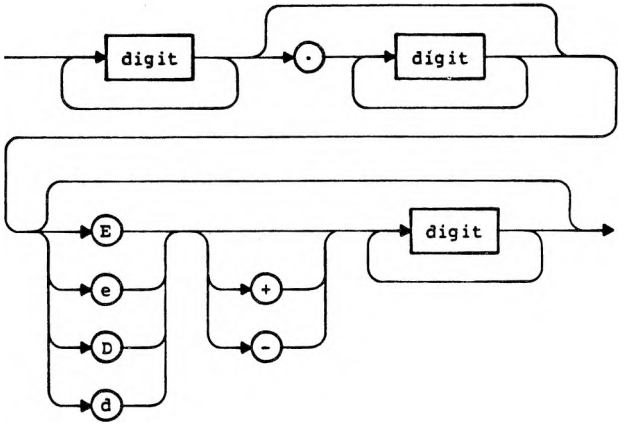
Factor



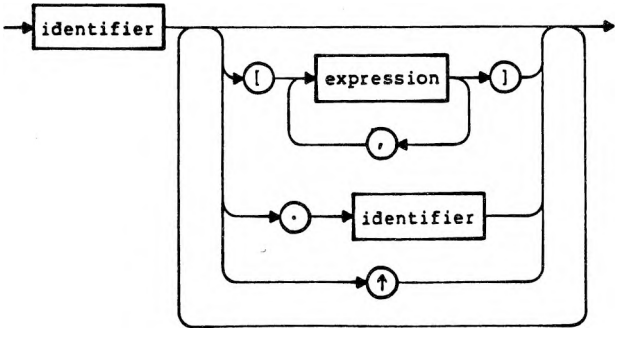
Cases



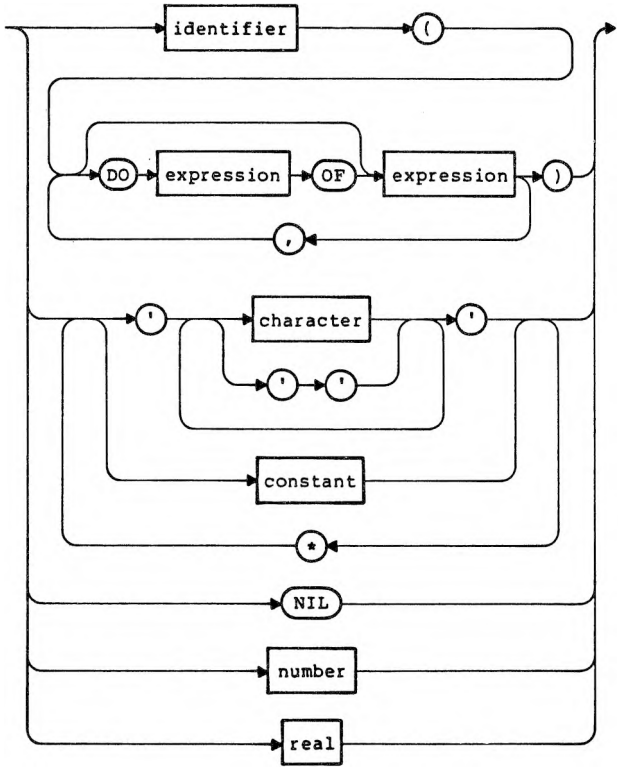
Real Number



Variable



Constant


identifier
(
DO
expression
OF
expression
)
,
'
character
'
'
'
constant
*
NIL
number
real

APPENDIX B: MS-PASCAL FEATURES AND THE ISO STANDARD

At this writing, the ISO Pascal standard, Level Ø and Level 1, is still in draft status. MS-Pascal generally conforms to this current draft standard, but does not yet implement the proposed conformant array mechanism. This controversial method of passing arrays of different bounds as one parameter type has not been tested, and the details change from draft to draft. The conformant array scheme is not part of the ANSI/IEEE standard nor the ISO Level Ø standard.

The super array type in MS-Pascal provides conformant array parameters, as well as dynamic length arrays allocated on the heap. Programs correctly written to the ISO standard (Level Ø) or to the ANSI/IEEE standard should run correctly, without changes, under MS-Pascal. However, since MS-Pascal features introduce new reserved words and other elements, this goal cannot be fully realized.

B.1 MS-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 that the fact that the error is not caught. These "errors not caught," and other differences between MS-Pascal and the ISO standard, are described below. An MS-Pascal program that conforms or tests conformance to the ISO standard must have both the metacommands $STANDARD and $DEBUG on.

MS-Pascal allows the following minor extensions to the current ISO/ANSI/IEEE standard:

- A question mark (?) as a substitute for the up arrow (^)

o The underscore (_) in identifiers

Due to the way the compiler binds identifiers, the new reserved words added at the extend and system levels cannot be used as identifiers at the standard level. A new directive, EXTERN, and new predeclared functions are standard in MS-Pascal.

The current differences between MS-Pascal at the standard level and the current ISO/ANSI/IEEE standard are summarized in the following pages.

1. The ISO standard requires a separator between numbers and identifiers or keywords.

   MS-Pascal in some cases doesn't require a separator between a number and an identifier or keyword, e.g., "100mod" is accepted as "100 mod" without error.

2. The ISO standard does not allow passing a component of a PACKED structure as a reference parameter.

   MS-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.

   MS-Pascal permits all 256 8-bit values as CHAR values; with some operating systems, a particular CHAR value (e.g., carriage return) is also the line marker character.

4. The ISO standard requires a variant to be given for all possible tag values.

   MS-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 MS-Pascal, using an identifier and then redeclaring it in the same scope is an error not caught. For example, the following has two meanings for Y in the same scope:

   CONST X=Y; VAR Y: CHAR;

   MS-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.

   MS-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 READ(LN) and 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 reading or 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 run-time.

   MS-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. 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.

   MS-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.

   MS-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).

    MS-Pascal does not catch these as errors.

11. The ISO standard currently defines I MOD J as an error if J < Ø and the result of MOD is positive, even if I is negative.

    MS-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.

    MS-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.

    MS-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. 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.

    MS-Pascal allows CHR taking any ordinal type.

B.2 SUMMARY OF MS-PASCAL FEATURES

This outline summarizes MS-Pascal extensions to the ISO standard. Unless otherwise noted, all are at the extend level.

1. Syntactic and Pragmatic Features

The metalanguage (standard level)

$BRAVE	$INTEGER	$PAGEIF	$SKIP
$DEBUG	$LINE	$PAGESIZE	$SPEED
$ENTRY	$LINESIZE	$POP	$STACKCK
$ERRORS	$LIST	$PUSH	$STANDARD
$EXTEND	$MATHCK	$RANGECK	$SUBTITLE
$GOTO	$MESSAGE	$REAL	$SYMTAB
$INCLUDE	$NILCK	$ROM	$SYSTEM
$INCONST	$OCODE	$RUNTIME	$TAGCK
$INDEXCK	$OPTBUG	$SIMPLE	$TITLE
$INTICK	$PAGE	$SIZE	$WARN

$IF $THEN $ELSE $END

Extra listing (standard level)

Flags for jumps, globals, identifier level, control level, header, trailer

Textual error and warning messages

Syntactic additions

! as comment to end of line

Square brackets equivalent to BEGIN/END

Nondecimal number notation

Numeric constants with _# or nn_# (where nn = 2..36)

DECODE/READ takes _# notation

ENCODE/WRITE with N of 2, 8, 10, 16

Extended CASE range

For CASE statements and record variants

OTHERWISE for all other values

A..B for range of values

2. Data types and modes

WORD type, WRD function, MAXWORD constant

REAL4 and REAL8 types

INTEGER4 type, MAXINT4 const;

FLOAT4, ROUND4, and TRUNC4 functions

Address types (system level)

ADR and ADS types and operators

VARS and CONSTS parameters

SUPER array types

Conformant parameters

Dynamic length heap variables

Multidimensional super arrays

STRING and LSTRING super types

LSTRING type, NULL constant, .LEN field

Explicit byte offsets in records (system level)

CONST and CONSTS reference parameters for constants and expressions

Structured (array, record, and set) constants
Extended functions returning any assignable type

Variable selection on values returned from functions

Attributes

EXTERN	PORT
EXTERNAL	PUBLIC
FORTRAN	PURE
INTERRUPT	READONLY
ORIGIN	STATIC

3. Operators and intrinsics

Extend level operators:

Shift operators: SHL SHR ISR

Bitwise logical: AND OR NOT XOR

Set operators: _< >

Constant expressions:

String constant concatenation with * operator

Numeric, ordinal, Boolean expressions in type clauses

Other constant functions:

CHR	LOWORD	WRD	<=
DIV	MOD	*	<>
HIBYTE	ORD	+	=
HIWORD	RETYPE	-	>
LOBYTE	SIZEOF	<	>=
LOWER	UPPER		

Additional intrinsic functions at extend level:

ABORT	EVAL	LOWORD
BYLONG	HIBYTE	RESULT
BYWORD	HIWORD	SIZEOF
DECODE	LOBYTE	UPPER
ENCODE	LOWER	

Additional intrinsic functions at system level:

FILLC	MOVESL
FILLSC	MOVESR
MOVEL	RETYPE
MOVER	

Intrinsic functions that operate on strings:

For STRING or LSTRING:

COPYSTR POSITN SCANEQ SCANNE

For LSTRING only:

CONCAT INSERT DELETE COPYLST

MS-FORTRAN REAL library functions (standard level)

MS-Pascal library functions (standard level):

ALLHQQ	ENDOQQ	LADDOK	PLYUQQ	TICS
BEGOQQ	ENDXQQ	LMULOK	PTYUQQ	TIME
BEGXQQ	ENABIN	LOCKED	RELEAS	UADDOK
DATE	FREECT	MARKAS	SADDOK	UMULOK
DISBIN	GTYUQQ	MEMAVL	SMULOK	UNLOCK
				VECTIN

4. Control flow and structure features

Control flow statements: BREAK, CYCLE, and RETURN

Sequential control operators: AND THEN and OR ELSE in IF, WHILE, REPEAT

Extended FOR loop: FOR VAR variable

VALUE section to initialize static variables

Mixed order LABEL, CONST, TYPE, VAR, VALUE sections

Compilable MODULES, with global attributes

UNIT INTERFACE and IMPLEMENTATION:

Interface version numbers, version checking

Optional rename of constituents

Guaranteed unique unit initialization

Optional unit initialization

5. Extend level input/output and files

Textfile line length declaration, TEXT (nnn)

READ enumerated, Boolean, pointer, STRING, LSTRING

WRITE enumerated, pointer, LSTRING

Negative M value to justify left instead of right

Temporary files

DIRECT mode files, SEEK procedure

ASSIGN, CLOSE, DISCARD, READSET, READFN procedures

FILEMODES type and constants, F.MODE access

Error trapping, F.TRAP and F.ERRS access

Enumerated I/O using identifier as string

6. System level I/O

Full FCBFQQ type equivalent to FILE types

APPENDIX C: MS-PASCAL AND OTHER PASCALS

At the standard level, MS-Pascal conforms to the current ISO draft standard. In theory, therefore, programs written in accordance with the ISO standard are portable and can be compiled with any MS-Pascal compiler with no problem. In practice, however, the majority of Pascal programs are written with at least some nonstandard features. In these cases, it is necessary to alter the Pascal source file to conform to the conventions used in MS-Pascal.

C.1 IMPLEMENTATIONS OF PASCAL

The areas in which different implementations of the Pascal language differ from one another fall into one of the following categories:

1. Interactive I/O

   MS-Pascal implements lazy evaluation to handle interactive I/O in a natural way. Other Pascals may implement this feature in different ways. For example, some systems require an initial READLN.

2. String handling

   MS-Pascal supports the super array type LSTRING to handle variable length strings efficiently. The ISO standard provides the PACK and UNPACK procedures for dealing with strings; other Pascals often have some improvement on the string handling facilities described in the standard.

3. Compiler controls

   Compiler controls implemented either as command line switches or as commands within source comments vary from Pascal to Pascal. To ensure portability, eliminate all embedded controls from comments.

4. Maximum set size

   The maximum set size varies from Pascal to Pascal. Some Pascals limit set size to 16 or 64 elements. In MS-Pascal, sets may contain up to 256 elements. This allows support of the SET OF CHAR.

5. Type compatibility

   The rules for type compatibility vary in their strictness. In some Pascals, structurally equivalent types with different names are compatible; in others (and in the ISO Standard), they are not.

6. Out of block GOTOs

   Some Pascals do not permit the out of block GOTOs that are permitted in MS-Pascal.

7. Heap management

   Rather than use the procedures NEW and DISPOSE for managing dynamic allocation of memory, some Pascals use the MARK and RELEASE procedures. MS-Pascal supports both methods. (MARKAS and RELEAS are the MS-Pascal names for MARK and RELEASE.)

8. OTHERWISE in CASE statements and variant records

   If OTHERWISE is omitted in a CASE statement, control does not automatically pass to the next

executable statement as in some other extended Pascals. Also, some other Pascals use the word ELSE or OTHERS instead of OTHERWISE.

9. Assigning filenames

The ASSIGN procedure in MS-PASCAL sets an operating system filename for a file. Some other Pascals use a second parameter to RESET and REWRITE for the filename.

10. Separate compilation

Most Pascals exclude the EXTERN (or EXTERNAL) directive for procedures and functions. Many support the idea of a MODULE and/or an INTERFACE and IMPLEMENTATION, although the syntax may differ. Some do not support PUBLIC and EXTERN variables, but may use a FORTRAN COMMON approach. In the latter case, for portability, you should give all global variables in one MS-Pascal VAR section, using [PUBLIC] in the PROGRAM and [EXTERN] in the MODULE, and $INCLUDE the same variable declarations in each.

11. Program parameters

Some Pascals ignore program parameters. In some Pascals, all files must be program parameters.

12. Procedural parameters

Several Pascals do not permit passing procedures and functions as parameters. Many do not permit passing any predeclared preocedures or functions.

C.2 MS-PASCAL AND UCSD PASCAL

Because UCSD Pascal is one of the more prevalent Pascals for microcomputers, conversion of source files from UCSD to MS-Pascal, and vice versa, is likely to be a common occurrence. This section discusses the differences and similarities between the two Pascals.

MS-Pascal has incorporated many of the UCSD extensions in one form or another. Table C-1 compares UCSD extensions with similar extensions available in MS-Pascal.

Table C-1: MS-Pascal and UCSD Pascal

UCSD EXTENSION	MS-PASCAL EQUIVALENT
ATAN	ARCTAN
BLOCKREAD	GETUQQ
BLOCKWRITE	PUTUQQ
CLOSE	CLOSE
CLOSE (F, LOCK)	CLOSE (F)
CLOSE (F, PURGE)	DISCARD (F)
CONCAT	CONCAT
COPY	COPYLST or MOVEL
DELETE	DELETE
EXIT	RETURN or GOTO
FILLCHAR	FILLC and FILLSC
HALT	ENDXQQ
INSERT	INSERT
IORESULT, $I	ERRS and TRAP fields
LENGTH	.LEN or STR [Ø]
LOG	LNDRQQ
MARK	MARKAS
MEMAVAIL	MEMAVL
MOVELEFT	MOVEL and MOVESL
MOVERIGHT	MOVER and MOVESR
POS	POSITN
RELEASE	RELEAS
SCAN	SCANEQ and SCANNE

SEEK	SEEK
SIZEOF	SIZEOF
STR	ENCODE
STRING [n]	LSTRING (n)
UNIT	UNIT
Untyped Files	FCBFQQ type

The following notes describe comparative points of interest.

1. The UCSD STRING [n] type is logically similar to the MS-Pascal LSTRING (n) type. Both contain the length of a variable length string in element zero of an ARRAY of CHAR.

2. UCSD Pascal allocates pointer variables on the heap with MARK and RELEASE. Other Pascals normally use NEW and DISPOSE. MS-Pascal permits both methods of dynamic memory allocation.

3. MS-Pascal units are like UCSD Pascal units, with the following exceptions. In MS-Pascal, an INTERFACE must appear first in any compiland using it. Since UCSD Pascal has its own special file system, the name of the unit can be used to find the interface filename in a standard way.

   MS-Pascal requires a list of all identifiers exported from the unit in the UNIT clause itself and makes it optional in a USES clause. Different identifiers may be given in a USES clause to avoid identifier conflicts.

   Finally, MS-Pascal provides for unit initialization code and interface version control. Neither of these are available in UCSD Pascal.

4. CONCAT is a function in UCSD Pascal; in MS-

Pascal, it is a procedure.

5. In UCSD Pascal, when a CASE statement whose control value does not select a statement is executed, the statement following the CASE statement is executed. In MS-Pascal, you must include an empty OTHERWISE clause to obtain this effect.

6. UCSD Pascal permits the use of the EOF (F) and EOLN (F) functions on a closed file; in MS-Pascal, this is an error.

7. UCSD Pascal permits comparison of records and arrays with the equal sign (=) and the not-equal sign (<>). In MS-Pascal, you must RETYPE the records and arrays to the same length STRING type, and then compare them as strings.

APPENDIX D: ASCII CHARACTER CODES

Dec	Hex	CHR	Dec	Hex	CHR
000	00H	NUL	031	1FH	US
001	01H	SOH	032	20H	SPACE
002	02H	STX	033	21H	!
003	03H	ETX	034	22H	"
004	04H	EOT	035	23H	#
005	05H	ENQ	036	24H	$
006	06H	ACK	037	25H	%
007	07H	BEL	038	26H	&
008	08H	BS	039	27H	'
009	09H	HT	040	28H	(
010	0AH	LF	041	29H	)
011	0BH	VT	042	2AH	*
012	0CH	FF	043	2BH	+
013	0DH	CR	044	2CH	,
014	0EH	SO	045	2DH	-
015	0FH	SI	046	2EH	.
016	10H	DLE	047	2FH	/
017	11H	DC1	048	30H	0
018	12H	DC2	049	31H	1
019	13H	DC3	050	32H	2
020	14H	DC4	051	33H	3
021	15H	NAK	052	34H	4
022	16H	SYN	053	35H	5
023	17H	ETB	054	36H	6
024	18H	CAN	055	37H	7
025	19H	EM	056	38H	8
026	1AH	SUB	057	39H	9
027	1BH	ESCAPE	058	3AH	:
028	1CH	FS	059	3BH	;
029	1DH	GS	060	3CH	<
030	1EH	RS	061	3DH	=

Dec	Hex	CHR	Dec	Hex	CHR
Ø62	3EH	>	Ø95	5FH	_
Ø63	3FH	?	Ø96	6ØH	`
Ø64	4ØH	@	Ø97	61H	a
Ø65	41H	A	Ø98	62H	b
Ø66	42H	B	Ø99	63H	c
Ø67	43H	C	1ØØ	64H	d
Ø68	44H	D	1Ø1	65H	e
Ø69	45H	E	1Ø2	66H	f
Ø7Ø	46H	F	1Ø3	67H	g
Ø71	47H	G	1Ø4	68H	h
Ø72	48H	H	1Ø5	69H	i
Ø73	49H	I	1Ø6	6AH	j
Ø74	4AH	J	1Ø7	6BH	k
Ø75	4BH	K	1Ø8	6CH	l
Ø76	4CH	L	1Ø9	6DH	m
Ø77	4DH	M	11Ø	6EH	n
Ø78	4EH	N	111	6FH	o
Ø79	4FH	O	112	7ØH	p
Ø8Ø	5ØH	P	113	71H	q
Ø81	51H	Q	114	72H	r
Ø82	52H	R	115	73H	s
Ø83	53H	S	116	74H	t
Ø84	54H	T	117	75H	u
Ø85	55H	U	118	76H	v
Ø86	56H	V	119	77H	w
Ø87	57H	W	12Ø	78H	x
Ø88	58H	X	121	79H	y
Ø89	59H	Y	122	7AH	z
Ø9Ø	5AH	Z	123	7BH	{
Ø91	5BH	[	124	7CH	|
Ø92	5CH	\	125	7DH	}
Ø93	5DH	]	126	7EH	~
Ø94	5EH	^	127	7FH	DEL

Dec=Decimal, Hex=Hexadecimal (H), CHR=Character,
LF=Line Feed, FF=Formfeed, CR=Carriage Return,
DEL=Rubout

APPENDIX E: SUMMARY OF MS-PASCAL RESERVED WORDS

Reserved words at the standard level

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

Additional reserved words at the extend level:

BREAK	INTERFACE	RETURN	USES
CONSTS	ISR	SHL	VALUE
CYCLE	MODULE	SHR	VARS
IMPLEMENTATION	OTHERWISE	UNIT	XOR

Additional reserved words at the system level:

ADR ADS

Names of attributes:

EXTERN	ORIGIN	PURE
EXTERNAL	PORT	READONLY
FORTRAN	PUBLIC	STATIC
INTERRUPT		

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. This provides compatibility with a number of other Pascals.

APPENDIX F: SUMMARY OF AVAILABLE PROCEDURES AND FUNCTIONS

Table F-1 provides a summary listing of all available functions and procedures, along with the name of the group in which they are presented in Section 14.1, "Categories of Available Procedures and Functions.

Table F-1: Available Procedures and Functions

NAME	DESCRIPTION	CATEGORY
ABORT	Terminate program	Extend 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
ALLHQQ	Allocate heap item	Library
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
A2DRQQ	REAL8 arc tangent function	Arithmetic
A2SRQQ	REAL4 arc tangent function	Arithmetic
BEGOQQ	Initialize user	Library
BEGXQQ	Overall initialization	Library
BYLONG	WORD or INTEGER to INTEGER4	Extend level
BYWORD	Put bytes in word	Extend level
CHDRQQ	REAL8 hyperbolic cosine	Arithmetic
CHR	Get ASCII char of value	Data conversion
CHSRQQ	REAL4 hyperbolic cosine	Arithmetic
CLOSE	Close file	File system
CONCAT	Concatenate LSTRING	String
COPYLST	Copy to LSTRING	String
COPYSTR	Copy to STRING	String
COS	Cosine function	Arithmetic
DATE	Date function	Library

DECODE	Decode LSTRING to variable	Extend level
DELETE	Remove portion of LSTRING	String
DISBIN	Disable interrupts	Library
DISCARD	Close and delete file	File system
DISPOSE	Dispose of heap item	Dynamic alloc
ENABIN	Enable interrupts	Library
ENCODE	Encode expression to LSTRING	Extend 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	Extend level
EXP	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
FREECT	Give count of free blocks	Library
GET	Get next file component	File system
GTYUQQ	Direct terminal input	Library
HIBYTE	Get high BYTE	Extend level
HIWORD	Get high WORD	Extend 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
LOBYTE	Get low BYTE	Extend level
LOCKED	Resource locked status	Library
LOWER	Get lower bound	Extend level
LOWORD	Get low WORD	Extend level
MARKAS	Mark heap bounds	Library
MEMAVL	Available memory	Library
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 alloc
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
PLYUQQ	Direct terminal end line	Library
POSITN	Find position of substring	String
PRED	Predecessor function	Data conversion
PRDRQQ	REAL8 to REAL8 power	Arithmetic
PRSRQQ	REAL4 to REAL4 power	Arithmetic
PTYUQQ	Direct terminal output	Library
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
RELEAS	Release heap space	Library
RETYPE	Force expression to type	System level
RESET	Ready file for read	File system
RESULT	Return result of function	Extend 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	Extend level
SMULOK	16-bit signed multiply check	Library
SQR	Square function	Arithmetic
SQRT	Square root function	Arithmetic
SUCC	Successor function	Data conversion
THDRQQ	REAL8 hyperbolic tangent	Arithmetic
THSRQQ	REAL4 hyperbolic tangent	Arithmetic
TICS	Time in arbitrary units	Library
TIME	Time of day function	Library
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
UNLOCK	Unlock resource	Library
UNPACK	Unpack STRING to array	Data conversion
UPPER	Get upper bound	Extend level
VECTIN	Set interrupt vector	Library
WRD	Convert to WORD value	Data conversion
WRITE	Write file	File system
WRITELN	Write line to file	File system

APPENDIX G: SUMMARY OF MICROSOFT PASCAL METACOMMANDS

Table G-1 provides a single alphabetical list of all of the metacommands diescribed in Chapter 17. Defaults, if any, are shown following metacommand in column one.

Table G-1: MS-Pascal Metacommands

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.
$EXTEND	Adds extend level features.
$GOTO-	Flags GOTOs as "considered harmful."
$IF <constant> $THEN <text1> $ELSE <text2> $END	Allows conditional compilation <text1> source if <constant> is greater than zero.
$INCLUDE:'<file>'	Switches compilation to file named.
$INCONST	Allows interactive setting of constant values at compiletime.

$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.
$STANDARD	Enables standard level only.
$SUBTITLE:'<subt>'	Sets page subtitle.
$SYMTAB+	Sends symbol table to source listing.
$SYSTEM	Adds extend and system level features.
$TAGCK-	Checks tag fields in variant records.
$TITLE:'<title>'	Gives page title for source listing.
$WARN+	Gives warning messages in source listing.

APPENDIX H: MESSAGES

This appendix lists all of the error numbers and messages you are likely to encounter while using the MS-Pascal compiler and run-time system. These error conditions fall into several categories:

1. Compile-time warnings

2. Compile-time errors caught

3. Compiler internal errors

4. Errors (both compiletime and run-time) defined by the ISO standard not caught in MS-Pascal

5. Run-time file system errors

6. Run-time non-file system errors caught only if the appropriate switch is on

7. Run-time non-file system errors always caught

Linker errors are specific to the linker for the operating system with which you are working and are therefore included in your MS-Pascal Compiler User's Guide.

Error conditions:

- o May go undetected.
- o May be detected by the compiler.
- o May be detected by the run-time system.

An error is "caught" if the compiler or run-time 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 and 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.

Compiletime errors include all 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 MS-Pascal. Generally, these are infrequent or very hard to detect conditions, not caught as errors in MS-Pascal, but which might be in another implementation.

H.1 COMPILER FRON 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 (#128, #129, and #130) 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 includes the filename and one of the following phrases:

HARD DATA	**E.g., check sum error.**
DISK FULL	**Disk is full.**
FILE ACCESS	**E.g., file not found.**
FILE SYSTEM	**Other or internal error.**

The front end may also get one of two compiler run-time errors:

Error: Compiler Out Of Memory

This usually occurs when too many identifiers have been declared. See Chapter 6 in your MS-Pascal Compiler User's Guide for suggestions on how to handle this situation.

Error: Compiler Internal Error

No matter what source program is compiled, this message should not appear. If it does, please report the condition to Microsoft 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 your dealer.

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.

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 for your operating system and has been truncated. See your MS-Pascal User's Guide for the current maximum.

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 Meta Command Expected Command Ignored

The compiler found a dollar sign ($) at the start of a comment, but not a metacommand identifier.

109 Unknown Meta Command Ignored

The compiler found a metacommand identifier that it didn't recognize or that is invalid in this version of MS-Pascal.

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.

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 Meta Command

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 Meta Command 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 not either 4 or 8 for $REAL:n or 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 Meta

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 Meta Command Skipped

The compiler found an $IF metacommand but no subsequent $THEN or $ELSE. The $IF command has been ignored.

123 Unexpected Meta Command Skipped

The compiler found a $THEN metacommand unrelated to any $IF metacommand. The $THEN command is ignored.

124 Unexpected Meta Command

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. But 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.

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 exceed the maximum of 65534 bytes.

135 Static Memory Overflow

Static memory allocation has exceed 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 should be in a statement, declaration, etc.

x147 : Assumed =

The compiler found a colon where there should have been an equals sign and preceded as if the correct symbol were present.

148 = Assumed :

The compiler found an equals sign where it expected a colon and proceeded as if the correct symbol were present.

149 := Assumed =

The compiler found colon followed by an equals sign where it expected an equals sign only and proceeded as if the correct symbol were present.

150 = Assumed :=

The compiler found an equals sign where it expected a colon following by an equals 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.

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.

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 equals sign where it expected one, but proceeded as if it were present.

166 Insert :=

The compiler didn't find a colon followed by an equals 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.

185 Invalid Symbol Begin Skip

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 #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.

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 INTERGER 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 ().

201 Types Not Assignment Compatible

You have attempted to use incompatible types in an assignment statement or value parameter. See Chapter * in the manual for type compatibility rules.

202 Types Not Compatible In Expression

You have attempted to mix incompatible types in an expression. See Chapter * in this manual 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, or concurrency, for binary files). 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 non-constant expression where it expected a constant ordinal expression.

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 exceed 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 non-constant value in a structured constant and assumed its value was zero.

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.

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 the ORD or WRD 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 not available in this version of MS-Pascal.

12. The ORD or WRD of an INTEGER4 value is out of range.

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.

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.

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 the attribute INTERRUPT, directly or indirectly. INTERRUPT does not allow this.

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 have 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 id) 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.

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.

272 Word Constant Overflow

A WORD constant minus a WORD constant has given a negative result.

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.

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.)

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.

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.

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.

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 id) 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.

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 return 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.

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 incorrectly-used INTERRUPT procedure, that is, one that has parameters or is a function.

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.

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 non-structured 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.)

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.

364 Contains File Initialize Module

The compiler found an unitialized 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 unitialized file variables 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.

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 Pascal 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.

H.2 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:

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 [Pcode only].

Try breaking up compiland into modules or units.

5 Range error (number too large to fit into target).

H.3 COMPILER INTERNAL ERRORS

All errors labeled "Compiler" in Section H.1 are compiler internal errors that should never occur. In the event that one does occur, report it to your dealer 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.

H.4 RUN-FILE SYSTEM ERRORS

File system error codes range from 1000 to 1999. Error codes go into the ERRC field of the file control block. Codes from 1000 to 1099 designate errors (from Unit U) that are specific to your operating system. Those from 1100 to 1199 identify Pascal file system errors (from unit F).

File system errors all have the format:

<error type> error in file <filename>

followed by the error code, and in some versions an error status, which is an operating system error return word. The <error type> field is based on the ERRS field of the file control block, as follows:

1 Hard Data

Hard data error (parity, CRC, checksum,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 temp names, etc.

8 Device Full

Disk full, directory full, all channels allocated.

9 Unknown Device

Unit/device/volume not found

10 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.

H.4.1 OPERATING SYSTEM RUN-TIME ERRORS

The following error messages are specific to particular operating systems.

1000 Write error when writing end of file
1001 Unknown device name

CP/M-80 and CP/M-86 only. Occurs when no filename has ever been assigned in a RESET or REWRITE.

1002 Filename extension with more than 3 characters

1003 Error during creation of new file

(disk or directory full)

1004 Error during open of existing file (file not found)

1005 Filename with more than 8 or zero characters

1006 Device cannot do input or output

(CP/M-80 and CP/M-86 only)

1007 Total filename length over 21 character

1008 Write error when advancing to next record

1009 File too big (over 65535 logical sectors)

1010 Write error when seeking to direct record

1011 Attempt to open a random file to a non-disk device

1012 Forward space or back space on a non-disk device

(FORTRAN error only)

1013 Disk or directory full error during forward space or back space

(FORTRAN error only)

%RNFCJL Can't justify line on output page H-35; on input line 15676 of page 1 of file "DSK:MCP.RNO"

H.4.2 MS-PASCAL FILE SYSTEM ERROR CODES (1100-1199)

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

H.5 OTHER RUN-TIME ERRORS (2000-2999)

Non-file 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.

H.5.1 MEMORY ERRORS (2000-2049)

%RNFILC Illegal command: ".Errors>memory"

on output page H-37; on input line 15747 of page 1 of file "DSK:MCP.RNO"

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.

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 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 Microsoft Corporation.

2031 NIL Pointer Reference

DISPOSE or $NILCK+ found a pointer with a NIL (i.e., Ø) value.

2032 Uninitialized Pointer

DISPOSE or $NILCK+ found an uninitialized (value 1) pointer. This only occurs if the metacommand $INITCK is on.

2033 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.)

2034 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.

2035 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.

H.5.2 ORDINAL ARITHMETIC ERRORS (2050-2099)

2050 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.

2051 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 #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.

2084 Integer Zero To Negative Power

There was an attempt to raise zero to a negative power (FORTRAN error only).

H.5.3 TYPE REAL ARITHMETIC ERRORS (2100-2149)

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.

2102 SIN or COS Argument Range

The parameter for a SIN or COS function is too large to yield a meaningful result. This error is only caught in 8080 systems.

2103 EXP Argument Range

The parameter for an EXP function is too large to yield a result that fits in representation. This error is only caught in 8080 systems.

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.Ø

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 Infinity (Unitialized Or Previous Error)

The REAL value called "infinity" was encountered. This may occur if the $INITCK metacommand is on and an unitialized REAL value is used, or if a previous error set a variable to indefinite as part of its masked error response.

2137 Missing Arithmetic Processor

You linked your program with the run-time library intended for use with the 8Ø87 numeric coprocessor, but there is no coprocessor on your system. Relink your program with the run-time library that emulates floating point arithmetic.

2138 REAL IEEE Denormal Detected

A very small real number was generated and may no longer be valid due to loss of significance.

2139 REAL Precision Loss

An arithmetic operation on the 8087 numeric coprocessor has generated a loss of numeric precision in the result of an operation.

2140 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.

H.5.4 STRUCTURED TYPE ERRORS (2150-2199)

2150 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.

2180 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.

H.5.5 INTEGER4 ERRORS (2200-2249)

2200 Long Integer Divide By Zero

An INTEGER4 value is divided by zero. This error is always caught.

2201 Long Integer Math Overflow

n INTEGER4 value is too large for representation. This error is always caught.

2234 Long Integer Zero To Negative Power

There was an attempt to raise zero to a negative power (FORTRAN error only).

H.5.6 OTHER ERRORS (2400-2999)

2400 Illegal Pcode

This is an internal error, which may occur in P-code systems only.

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.