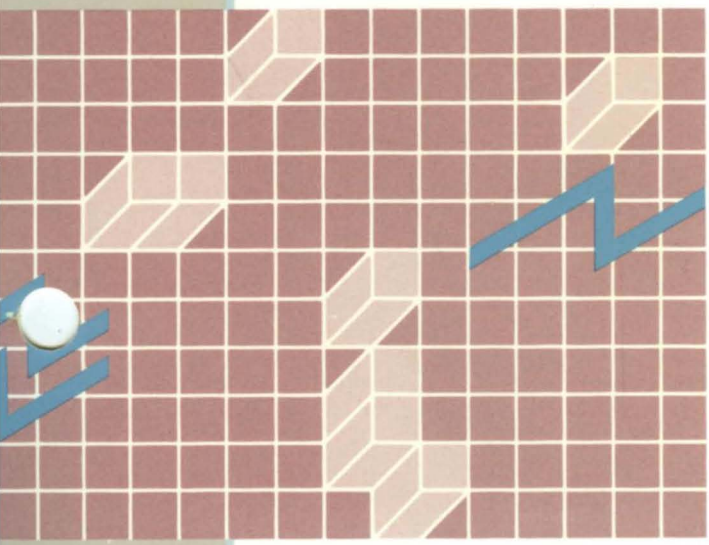
# *Pascal Reference*

## *Volume 2*



# PASCAL REFERENCE MANUAL: VOLUME 2

Specifications Subject to Change.



**Third Edition (September 1984) A-09-00868-01-A**

## LIST OF FIGURES



## LIST OF TABLES

# 13 INTRODUCTION TO PROCEDURES AND FUNCTIONS

Procedures and functions are both subprograms (subroutines) that execute under the supervision of a main program. A procedure is a subprogram that is invoked as a program statement. A function is the same as a procedure, except that it returns a value and is invoked as an expression instead of as a statement.

Unlike programs, procedures and functions can be nested within each other and can even call themselves. Furthermore, they have sophisticated parameter passing capabilities that programs lack.

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. The declarations and body together are called the block.

In a program text, procedures and functions are declared before the main body of the program, and before any other procedures or functions that call them. They can alternatively be defined as EXTERN and declared in a different module; the reference is then resolved later by the Linker. 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.

Example of a procedure declaration:

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

{Beginning of declaration section}
LABEL 123
CONST ATOP = 199;
TYPE  INDEX = Ø..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 := Ø 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;
```

## PROCEDURES

The foregoing example illustrates the general format of a procedure declaration. The heading is followed by:

- declarations for labels, constants, types, variables, and values
- local procedures and functions
- 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 **extend level** you can use the RETURN statement to exit the current procedure, function, program, or implementation. (See Section 12, "Statements," for a discussion of how to use the RETURN statement.)

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.

At the standard level, declarations must appear in the following order:

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, in practice, it is clearer to give all data declarations first. However, if you put variable declarations after procedure and function declarations you can guarantee that these variables will not be used by any of the procedures or functions.

In general, the initial values of variables are not defined. However, you can use the VALUE section to explicitly initialize program, module, implementation, STATIC, and PUBLIC variables. The VALUE section, an extension offered by this version of Pascal, should follow the VAR section. If the initialization switch ($INITCK) is on, all INTEGER, INTEGER subrange, REAL and pointer variables are set to an uninitialized value. File variables are always initialized, regardless of the setting of the initialization switch.

## FUNCTIONS

Like procedures, functions are subprograms. A function, however, is invoked in an expression, instead of a statement. Also, a function returns a single value.

A function declaration defines the parts of a program that compute a value. A function is 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;
{Returns an INTEGER value}
```

At the standard level, functions can return a pointer or any simple type (ordinal, REAL, or INTEGER4.)

At the **extend level**, functions can return any simple, structured, or reference type. However, they cannot return any type that cannot be assigned (that is, a super array type or a structure containing a file.) A super array derived type is permitted, however.

A function identifier on the left side of an assignment within the function body or the body of its internal procedure or function does not invoke the function recursively. Instead, it refers to the function's local variable, which contains its current value. The local variable is created by the compiler, not declared in the VAR section by the programmer. On return from the function, the value of the local variable is returned.

Using the function identifier in one of the following places in the function block gets the address of the local variable:

- a reference parameter
- the record of a WITH statement
- the operand of an ADR or ADS operator

A function identifier used in an expression within the body of the function invokes the function recursively, rather than giving the current value of the function.

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 does not generate code to check for this assignment unless the initialization switch ($INITCK) 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 during compilation.

To obtain the current value of the function within an expression within its block, use the RESULT function. RESULT takes the function identifier as a parameter and is available at the **extend level.**

The following is an example of the 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.

Instead of using the function's local variable, you can invoke the function recursively and use the return value.

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 {On Tuesdays we recurse}
    BEGIN
      IF B = Ø THEN
           BEGIN SUM := A;
           RETURN END;
      WITH (SUM (A,B-1)){Call SUM recursively}
           DO SUM.I := I + 1
             {I is result of call}
    END
    ELSE {Use function's local variable}
      WITH SUM
           DO I := A + B {I is local variable}
END;
```

At the **extend level** you can use the RETURN statement to exit the current procedure, function, program, or implementation. (See Section 12, "Statements," for a discussion of how to use the RETURN statement.)

## PARAMETERS TO PROCEDURES AND FUNCTIONS

Procedures and functions can take three different types of parameters:

- Value parameters, which pass an actual value.
- Reference parameters, which pass the address of a variable.
- Procedural and functional parameters, which pass a procedure or function.

Each of these is discussed separately 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, by specifying an identifier in the heading. When the function or procedure is called, an actual parameter substitutes for the formal parameter given earlier and takes the form of a variable, value, or expression.

**Extend level** Pascal has several parameter features:

- A super array type can be passed as a reference parameter.
- A reference parameter can be declared READONLY.
- Explicit segmented reference parameters can be declared.

### VALUE PARAMETERS

When a value parameter is passed, the actual parameter is an expression. That expression is evaluated in the scope of the calling procedure or function and the value is 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 value parameters}
  .
  .
  X := ADD (Y, ADD(1.11, 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 calls the function ADD to evaluate an actual parameter.)

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 requires a run-time error check if the range-checking switch ($RANGECK) is on. In addition, SET and LSTRING value parameters may require extra generated code for size adjustment.

A file variable or super array variable cannot be passed as a value parameter, since it cannot be assigned. However, a variable with a type derived from a super array or a file buffer variable can be a value parameter. File buffer variables are then evaluated as they would be in an expression.

**REFERENCE PARAMETERS**

When a reference parameter is passed at the standard level, the keyword VAR precedes the formal parameter. In addition, the actual parameter must be a variable, not an expression.

The formal parameter denotes this actual variable during the execution of the procedure. Any operation on the formal parameter is performed immediately on the actual parameter, by passing the machine address of the actual variable to the procedure. This address is an offset into the default data segment.

Example of reference 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. Note that, 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 actual parameter 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 the subsection "Listing File Format," in Section 17, "Metacommands" for information about how to interpret the listing file characters.)

None of the following can be passed as VAR parameters:

- components of PACKED structures (except CHAR of a STRING or LSTRING)
- variables with READONLY or PORT attributes, including CONST and CONSTS parameters and the FOR control variable

Passing a file buffer variable by reference generates a warning message, because it bypasses the normal file system call generated by the use of any buffer variable. These calls are not generated when a file variable is passed by reference.

A VAR parameter passes an address that is really an offset into a default data segment. In some cases, access to objects residing in other segments is required. To pass these objects by reference, you must tell the compiler to use a

segmented address containing both segment register and offset values. To do so, you can use the **extend level** parameter prefix VARS instead of VAR:

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

You can only use VARS as a data parameter in procedures and functions, not in the declaration section of programs, procedures, or functions.

### Super Array Parameters

Super array parameters can 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 itself is a reference parameter of the super array type.

The actual parameter type must be a type derived from the super array type or the super array type itself (that is, another reference parameter or dereferenced pointer.) Except for comparing LSTRINGS, super array type parameters cannot be assigned or compared as a whole.

The actual upper and lower bounds of the array are available with the UPPER and LOWER functions: this permits routines that can operate on arrays of any size. An LSTRING actual parameter can be passed to a reference parameter of the super array type STRING. Therefore, the super array parameter STRING can be used for procedures and functions that operate on strings of both STRING and LSTRING types.

Example of super array parameters:

```
TYPE REALS = ARRAY [Ø..*] OF REAL;

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

(For more information, see the subsections "Super Arrays," "STRINGs," and "LSTRINGs" in Section 6, "Arrays, Records, and Sets.")

**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 can be constants, since it eliminates the need for a time-consuming value parameter copy. The actual parameter can be a variable, function result, or constant value.

No assignments can be made to the CONST parameter or any of its components. CONST super array types are permitted. A CONST parameter in one procedure cannot be passed as a VAR parameter to another procedure. However, it is permissible to pass a VAR parameter in one procedure as a CONST parameter in another.

Example of a CONST parameter:

```
PROCEDURE ERROR (CONST ERRMSG: STRING):
```

A CONST parameter is passed as 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 the segment address and the offset values.

The **extend level** includes the parameter prefix CONSTS. 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 only be used as a data parameter in procedures and functions, not in the declaration section of programs, procedures, and functions.

You can also pass the value of an expression as a CONST or CONSTS parameter. The expression is evaluated and assigned to a temporary (hidden) variable in the frame of the calling procedure or function. You should enclose such an expression in parentheses to force its evaluation.

A function identifier can be passed by reference as a VAR, VARS, CONST, or CONSTS parameter. The function's local variable is passed, so the call must occur in the function's body or in a procedure or function declared within 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;
```

**PROCEDURAL AND FUNCTIONAL PARAMETERS**

Procedural parameters can be useful in the following circumstances:

- in numerical analysis
- in calling some library routines
- in special applications

In numerical analysis, for example, 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_DOOR_AJAR;
     (VAR DD : DOOR; SS : SPEED; DD :
               DIRECTION);
```

All the procedures in the example have parameter lists of equal length. The types of 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 := 0;
    MOVE_DOOR (XX, YY, ZZ);
    {One of the three procedures declared}
    {previously is executed here.}

    IF XX = BARN AND ZZ = SHUT
      THEN DOOR_STATUS := 1;
```

```
        IF XX = CELL AND ZZ = OPEN
        THEN DOOR_STATUS := 2;

        IF XX = DOG_HOUSE AND ZZ = SHUT
        THEN DOOR_STATUS := 3
END;
```

Use of the procedural parameter MOVEDOOR might occur in program statements as follows:

```
IF DOOR_STATUS
  (SLAM_DOOR, CELL, FAST, SHUT) = Ø
THEN
  SOCIETY := SAFE;
IF DOOR_STATUS
  (OPEN_DOOR_WIDE, BARN, SLOW, OPEN) = Ø
THEN
  COWS_ARE_OUT := TRUE;
IF DOOR_STATUS
  (LEAVE_AJAR, DOG_HOUSE, SLOW, OPEN) = Ø
THEN
  DOG_CAN_GET_IN := TRUE;
```

In each case above, the actual procedure list is compatible with the formal list, both in the number and in the types 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.

A procedure or function passed as a parameter to an EXTERN procedure or function must itself be PUBLIC or EXTERN. The procedural parameter must also be declared PUBLIC in the external procedure declaration. If they are nested, they must be declared at the lowest nesting level.

You cannot pass predeclared procedures and functions compiled as inline code; you can only call them in passed subroutines. Also, the READ, WRITE, ENCODE, and DECODE families are translated

into other calls by the compiler, based on the argument types, and so cannot be passed. Corresponding routines in the file unit or encode/decode unit can be passed, however. For example, a READ of an INTEGER becomes a call to RTIFQQ and this procedure can be passed as a parameter.

The following intrinsic procedures and functions cannot be passed as procedure or function parameters:

- at the standard level

| | | | |
|---|---|---|---|
| ABS | EOLN | PACK | SQR |
| ARCTAN | EXP | PAGE | SQRT |
| CHR | LN | PRED | SUCC |
| COS | NEW | READ | UNPACK |
| DISPOSE | ODD | READLN | WRITE |
| EOF | ORD | SIN | WRITELN |

- at the **extend level**

| | | | |
|---|---|---|---|
| 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 procedural parameter 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
          GLOB := Ø;
          IF I = Ø
            THEN BEGIN
            I := 1; XPR; BETA (GAMMA)
            END
          ELSE BEGIN
            GLOB := GLOB + 1;XPR
            END
        END;

  BEGIN
        I := Ø;
        BETA (DELTA)
  END;
```

The following list describes what happens in this example when ALPHA is called:

- BETA is called, passing the procedure DELTA.
- This latter call creates an instance of GLOB on the stack (call it GLOB1).
- BETA first clears GLOB1 by setting it to zero. Then, since I is Ø, the THEN clause is executed, which sets I to one and executes XPR, which is bound to DELTA.
- Therefore, 'Delta done' is written to OUTPUT.
- Now BETA is called recursively. BETA is passed GAMMA, and, at this time, the access path to any nonlocal variables used by GAMMA (for instance, GLOB1) is passed as well.
- The second call to BETA creates another instance of GLOB (GLOB2). When GLOB2 is cleared this time, I is 1, so GLOB2 is incremented.
- The 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.
- GAMMA returns, the second BETA call returns, the first BETA call returns, and finally, ALPHA returns.

## DIRECTIVES AND ATTRIBUTES

A directive gives information about the location of a procedure or function. A directive replaces the block of the procedure or function (declarations and body) and indicates that only the heading of the procedure or function occurs. Directives are available in standard Pascal. EXTERN and FORWARD are the only directives available. EXTERN can only be used with procedures or functions directly nested in a program, module, implementation, or interface. This restriction prevents them from accessing nonlocal stack variables.

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

Table 13-1 displays the directives and attributes that apply to procedures and functions, and the sections below describe them in detail.

The following rules apply when you combine attributes in the declaration of procedures and functions:

- A function can be given the PURE attribute.
- 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. (Modules and units are discussed in Section 16, "Compilable Parts of a Program.")
- PUBLIC and EXTERN are mutually exclusive, as are PUBLIC and ORIGIN.

**Table 13-1. Directives and Attributes for Procedures and Functions.**

| Name | Purpose |
|---|---|
| **Directive**[a] | |
| FORWARD | Lets you call a procedure or function before you give its block in the source file. |
| EXTERN | Indicates that a procedure or function resides in another module. |
| **Attribute**[b] | |
| PUBLIC | Indicates that a procedure or function can be accessed by other modules. |
| ORIGIN | Tells the compiler where the code for an EXTERN procedure or function resides. |
| INTERRUPT | Gives a procedure a special calling sequence that saves program status on the stack. |
| PURE | Signifies that the function does not modify any global variables. |

[a] Available at the standard level

[b] Available at the **extend level**

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 that applies 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. You can suppress the default PUBLIC attribute for each module by including empty attribute brackets ([]). Then, individual items can be declared PUBLIC or not within the module.

The PUBLIC attribute allows a procedure or function to be called from other compilands (that is, separately compiled parts of the program) and cannot be used with the EXTERN directive. The EXTERN directive permits a call to a procedure or function declared in another compiland. PUBLIC, EXTERN, and ORIGIN provide a low level way to link Pascal routines with other Pascal routines or routines in other languages.

A procedure or function declaration with the EXTERN or FORWARD directive consists only of the heading, without the block. EXTERN routines have an implied block outside of the program. FORWARD routines are FULLY DECLARED (that is, they have a block) later in the same compiland. Both directives are available at the standard level of Pascal. The keyword EXTERNAL is a synonym for EXTERN.

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

## THE FORWARD DIRECTIVE

A FORWARD directive 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. The procedure or function is actually declared later, without repeating the formal parameter list, the attributes, or the return type of a function.

Example of a FORWARD directive

```
{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 or attributes}
FUNCTION ALPHA;
BEGIN
  ALPHA := (Q = R);
  IF R < Ø.Ø THEN BETA (3.14, ALPHA)
END;
```

## THE EXTERN DIRECTIVE

The EXTERN directive identifies a procedure or function that resides in another loaded module. You give only the heading of the procedure or function, followed by the word EXTERN. The actual declaration (with the body) 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 Pascal programs by linking separately compiled units. See Section 16, "Compilable Parts of a Program."

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

```
FUNCTION POWER (X,Y: REAL): REAL; EXTERN;

PROCEDURE ACCESS (KEY: KYTP)  [ORIGIN SYSB+4];
                  EXTERN;
```

In these examples, the function POWER is declared EXTERN, as is the procedure ACCESS. Both are declared and defined in external compilands. ACCESS also has the ORIGIN attribute, which is discussed below in the subsection "The ORIGIN Attribute."

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

### THE PUBLIC ATTRIBUTE

The PUBLIC attribute indicates a procedure or function that you can access from other compilands. In general, you access PUBLIC procedures and functions from other compilands by declaring them EXTERN in the modules that call them. Thus, you declare a procedure PUBLIC and define it in one module, then use it in another simply by declaring it EXTERN.

As with PUBLIC variables, the names of PUBLIC procedures and functions are included in the symbol file produced by the Linker.

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 routines is to link separately compiled units. See Section 16, "Compilable Parts of a Program" 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: KYTP)
[ORIGIN SYSB+4, PUBLIC];
BEGIN
.
.
END;
{Illegal since ORIGIN must also be EXTERN.}
```

**THE ORIGIN ATTRIBUTE**

The ORIGIN attribute can only 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]; EXTERN;

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

In the first example, the procedure OPSYS begins at the absolute decimal address 0:8 and is declared EXTERN.

The value of the ORIGIN (#100 in the second example) can be any constant expression (composed of constants and identifiers of constants.)

In the second example, the function A_TO_D takes a DINT value as a parameter (SINT is the predeclared integer subrange from -127 to +127). The function is located at the hexadecimal address 100 (0:100).

As with the 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. Nor can you 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, for example:

```
PROCEDURE OPSYS [ORIGIN 2:8]; EXTERN;
```

A nonsegmented procedural origin assumes the current code segment with the offset given with the attribute.

### 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 to be 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 only be invoked when

the interrupt associated with it occurs. Furthermore, INTERRUPT procedures take no parameters. (To associate an INTERRUPT procedure with an interrupt see Section 23, "Interrupt Handlers" in the CTOS Operating System Manual.)

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

- a special calling sequence saves all status on the stack
- the status saved includes machine registers and flags, plus any special global compiler data such as the frame pointer
- 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 and are neither enabled or disabled by an INTERRUPT procedure.

This version of Pascal does not provide interrupt vectoring or enabling.

An INTERRUPT procedure should usually return normally, in order to continue processing in the interrupted routine. Therefore,

- You should not execute a GOTO that leaves an INTERRUPT procedure.
- All debug checking should be turned off (that is, $DEBUG-, $ENTRY-, and $RUNTIME-).
- Stack overflow cannot be checked even if $STACKCK is on.

The use of INTERRUPT procedures introduces reentrancy into Pascal code: generated code is reentrant, as is the run-time system (except for the heap unit and portions of the file unit.

Note that caution should be used when nonreentrant code is used in INTERRUPT procedures. 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.

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 (that is, 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 in the process of performing I/O when the interrupt occurs.

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

Functions are not considered PURE unless given the attribute explicitly. A PURE function should not

- assign to a nonlocal variable
- have any VAR or VARS parameters (CONST and CONST parameters are permitted)
- call any functions that are not PURE

- Use the value of a global variable.
- Modify the referents of references passed by value (for example, pointer or address type referents.)
- Do input or output.

Note, however, that the compiler does not check for the restrictions listed above.

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);
```

## 14 AVAILABLE PROCEDURES AND FUNCTIONS

All versions of Pascal predeclare a large number of common procedures and functions, which you do not have to declare in a program. Since predeclared procedures and functions are defined in a scope "outside" the program, you can redefine these identifiers within your program if you wish.

Library procedures and functions are also available. To use these you must declare them as external to your program (EXTERN).

Available procedures and functions implemented by our version of Pascal can be divided into two types:

- Those that are predeclared.
- Those that are not predeclared but are a part of the run-time library. These procedures and functions must be declared explicitly.

To promote portability, some of the predeclared procedures and functions for this version of Pascal are available only at the extend level.

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

Following is a description of each of the categories shown in the Table 14-1 and a list of the procedures and functions that each category includes.

Under the heading "Directory of Functions and Procedures," at the end of this section, you will find a detailed alphabetical directory of all the available procedures and functions. The entry for each procedure or function in this directory includes the syntax and a description, plus examples and notes as appropriate.

**Table 14-1. Categories of Available Procedures and Functions.**

| Category | Purpose |
|---|---|
| File system | Operate on files of different modes and structures |
| Dynamic allocation | Dynamically allocate and deallocate memory at run time |
| Data conversion | Convert data from one type to another |
| Arithmetic | Perform common transcendental and other numeric functions |
| String intrinsics | Operate on STRING and LSTRING type data |
| INTEGER/WORD Conversion | Compose and decompose one-byte, two-byte, and four-byte items |
| Expression evaluation | Provide various procedures for use in evaluating functions |
| Initialization, termination, and error routines | Provide initialization, termination, and error handling |
| I/O routines | Provide direct I/O to and from keyboard and video |
| Semaphore routines | Ensure exclusive access to a resource in a concurrent system |

## FILE SYSTEM

The 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 | Procedure | Function |
|---|---|---|
| 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 Section 15, "File-Oriented Procedures and Functions."

## DYNAMIC ALLOCATION

Two memory areas are available for Pascal programs, the short and long heap. The short heap is at most 64K bytes long, whereas the long heap can be longer.

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

Library heap management routines, which complement the standard NEW and DISPOSE procedures include:

- ALLHQQ — Returns the pointer value for an allocated variable with the size requested
- FREECT — Returns an estimate of how many times NEW can be called to allocate heap variables
- MARKAS — Marks the upper and lower limits of the heap
- MEMAVL — Returns the number of bytes available between the stack and the heap
- RELEAS — Disposes of heap space past the area set with a previous MARKAS call

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

At the **extend level**, the intrinsic function SIZEOF determines the current size of a variable.

A Pascal program can allocate and deallocate memory from the long heap using the functions described below. (Naturally, to access data in the long heap, the user must specify both the segment and the offset addresses, that is, the data are accessed using ADS type variables.) If, at allocation request, not enough memory is available from the long heap, memory from the short heap is allocated.

- ALLMQQ — Allocates a block of not more than 64K bytes on the long heap and returns the block address
- FREMQQ — Frees a memory block from the long heap; returns Ø if no errors are encountered, nonzero otherwise
- GETMQQ — Performs ALLMQQ and provides additional error checking; terminates the program and returns an error message

- DISMQQ — Performs FREMQQ with additional error checking; terminates the program and returns an error message

Two functions are available for preallocating memory space.

- PREALLOCHEAP
  Lets you specify how much storage to be allocated for the short heap. You can then use short lived memory without worrying about overlapping memory with the heap.

- PREALLOCLONGHEAP
  Preallocates the short-lived memory for the long heap. If PREALLOCLONGHEAP has not been called by the user, the first call to a long heap allocation routine allocates as much short-lived memory as possible for the short heap and take all the rest of the short-lived memory for the long heap (to satisfy the current and possible future requests). To avoid all the rest of the short-lived memory being allocated for the long heap, you can use PREALLOCLONGHEAP.

## DATA CONVERSION

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

CHR
FLOAT
FLOAT4
ODD
ORD
PACK
RETYPE
TRUNC
TRUNC4
UNPACK
WRD

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

- CHR (ordinal) to CHAR
- ORD (ordinal) to INTEGER
- WRD (ordinal) to WORD

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

- FLOAT Converts INTEGER to REAL
- FLOAT4 Converts INTEGER4 to REAL
- TRUNC Converts REAL to INTEGER
- TRUNC4 Converts REAL to INTEGER4
- ROUND Rounds REAL to INTEGER
- ROUND4 Rounds REAL to INTEGER4

PACK and UNPACK transfer components between packed and unpacked arrays. (Note, however, that in our version of Pascal, packed and unpacked arrays have the same format.)

ODD tests to see if the ordinal value of a variable is odd.

At the **extend level**, the RETYPE function changes the type of an expression arbitrarily.

## ARITHMETIC FUNCTIONS

All arithmetic functions take a CONSTS parameter of type REAL4 or REAL8 or a type compatible with INTEGER. 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 ($MATHCK), 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 in case of an error is undefined.

Table 14-3 lists the arithmetic function available, along with the run-time routine calls generated by the compiler 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 | Arc tangent | ATSRQQ | ATDRQQ |
| COS | Cosine | CNSRQQ | CNDRQQ |
| EXP | Exponential | EXSRQQ | EXDRQQ |
| LN | Natural log | LNSRQQ | LNDRQQ |
| SIN | Sine | SNSRQQ | SNDRQQ |
| SQR | Square | SNSRQQ | SNDRQQ |
| SQRT | Square root | SRSRQQ | SRDRQQ |

PRED and SUCC are arithmetic functions that operate on ordinal types. They determine the ordinal predecessor or successor to a variable, respectively. PRED AND SUCC are not predeclared.

The following no-overflow arithmetic routines are not predeclared, but are available to you in the run-time library. You must declare them, with the EXTERN directive, before using them in a program.

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

The 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 Run-time Library.**

| Operation | REAL4 | REAL8 |
|---|---|---|
| Arc cosine | ACSRQQ | ACDRQQ |
| Integral trunc | AISRQQ | AIDRQQ |
| Integral round | ANSRQQ | ANDRQQ |
| Arc sine | ASSRQQ | ASDRQQ |
| Arc tangent 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 implement 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 Pascal to do the same thing. The PURE attribute is useful to obtain more efficient code when you define such functions. 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;
```

## STRING INTRINSICS

The following intrinsics are available for use with STRINGs and LSTRINGs at the standard level:

- o CONCAT — Concatenates strings
- o DELETE — Deletes a specified number of characters from an LSTRING
- o INSERT — Inserts a STRING into an LSTRING
- o COPYLST — Copies a STRING to an LSTRING
- o COPYSTR — Copies a STRING to another STRING
- o POSITN — Returns the position of a pattern within a STRING
- o SCANEQ — Searches a STRING for a pattern and returns the number of characters skipped before the pattern is found
- o SCANNE — Operates like SCANEQ, except it stops scanning when a character not equal to the specified pattern is found

At the **extend level**, the following string intrinsics are also available:

- o FILLC — Fills a specified memory region with a specified number of copies of one character
- o FILLSC — Fills a specified memory region with a specified number of copies of one character
- o MOVEL — Starting at the lowest addressed byte of an array, moves a specified number of bytes
- o MOVER — Starting at the highest addressed byte of an array, moves a specified number of bytes
- o MOVESL — Like MOVEL, but operates with ADSMEM
- o MOVESR — Like MOVER, but operates with ADSMEM

The **extend level** intrinsics ENCODE and DECODE convert between internal and string forms of variables.

## INTEGER/WORD CONVERSION PROCEDURES

At the **extend level**, the following intrinsic procedures and functions are available to compose and decompose one-byte, two-byte, and four-byte items.

- HIBYTE — Returns the most significant byte of an INTEGER or WORD
- LOBYTE — Returns the least significant byte of an INTEGER or WORD
- BYWORD — Forms a WORD from two byte values
- HIWORD — Returns the high order word of the four bytes of the INTEGER4
- LOWORD — Returns the low order word of the four bytes of the INTEGER4
- BYLONG — Forms an INTEGER4 from two WORD or INTEGER values

## EXPRESSION EVALUATION

At the **extend level**, the following intrinsic procedures and functions are available for determining current value of expressions:

- EVAL
- LOWER
- UPPER

In addition, RESULT, another extend level intrinsic, determines the current value of a function.

## INITIALIZATION, TERMINATION, AND ERROR ROUTINES

BEGOQQ and ENDOQQ are called during initialization and termination, respectively. BEGOQQ and ENDOQQ are empty procedures. You can write your own BEGOQQ or ENDOQQ, for example, to invoke a

debugger or to write customized messages, such as the time of execution, to the video display. If you write you own, you must declare them [PUBLIC] and include the name in the "Object Modules" field of the command form when you link the program.

BEGXQQ can be called to restart a program and ENDXQQ to terminate it.

At the **extend level**, the intrinsic procedure ABORT invokes a run-time error.

## I/O ROUTINES

The following library routines support direct input to and output to and from the keyboard or video display.

- GTYUQQ — Reads a specified number of characters from the keyboard and stores them in memory
- PTYUQQ — Writes characters from memory to the video display
- PLYUQQ — Writes a linefeed character to the video display

These routines must be declared EXTERN when used.

## SEMAPHORE ROUTINES

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

## DIRECTORY OF PROCEDURES AND FUNCTIONS

This subsection contains a list of all available procedures and functions, both those that are predeclared and those library routines that can be used if declared as external (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.

### ABORT

```
PROCEDURE ABORT (CONST MESS : STRING; ERR1, ERR2 :
WORD);
```

An **extend level** intrinsic procedure.

Halts program execution in the same way as an internal run-time error. The STRING is an error message. The string parameter is a CONST, not a CONSTS parameter. The first WORD is an error code. (See Appendix H, "Messages," for error code allocations.) ERR2, which can be anything, returns 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, 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 only call other $RUNTIME routines.

**ABS**

```
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.

**ACSRQQ and ACDRQQ**

```
FUNCTION ACSRQQ (CONSTS A: REAL4): REAL4; EXTERN;
FUNCTION ACDRQQ (CONSTS A: REAL8): REAL8; EXTERN;
```

Arithmetic functions.

Return the arc cosine of A. Both A and the return value are of type REAL4 or REAL8, as shown.

These functions are from the run-time library and must be declared EXTERN before use.

**AISRQQ and AIDRQQ**

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

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 run-time library and must be declared EXTERN before use.

**ALLHQQ**

```
FUNCTION ALLHQQ (SIZE: WORD): WORD; EXTERN;
```

A library routine (heap management function).

Returns zero if the heap is full, one if the heap structure is in error, MAXWORD if the allocator has been interrupted. Otherwise, it returns the pointer value for an allocated variable with the size requested.

Generally, ALLHQQ is used 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.}
```

**ALLMQQ**

```
FUNCTION ALLMQQ(wants: WORD) : ADSMEM; EXTERN;
```

Allocates a block of 'wants' bytes on the long heap and returns the block address. The block cannot be larger than 64K bytes.

This function is from the run-time library and must be declared EXTERN before use.

**ANSRQQ and ANDRQQ**

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

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 run-time library and must be declared EXTERN before use.

**ARCTAN**

```
FUNCTION ARCTAN (X: REAL): REAL;
```

An arithmetic function.

Returns the arc tangent 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.

**ASSRQQ and ASDRQQ**

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

Arithmetic functions.

Return the arc sine of A. Both A and the return value are of type REAL8 or REAL4, as shown.

These functions are from the run-time library and must be declared EXTERN before use.

**ASSIGN**

```
PROCEDURE ASSIGN (VAR F : FILE OF..; 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 the subsection "Extend Level Procedures" in Section 15, "File-Oriented Procedures and Functions," for a description.

**ATSRQQ and ATDRQQ**

```
FUNCTION ATSRQQ (CONSTS A: REAL4): REAL4; EXTERN;
FUNCTION ATDRQQ (CONSTS A: REAL8): REAL8; EXTERN;
```

See ARCTAN.

**A2SRQQ and A2DRQQ**

```
FUNCTION A2SRQQ (A, B: REAL4): REAL4; EXTERN;
FUNCTION A2DRQQ (A, B: REAL8): REAL8; EXTERN;
```

Arithmetic functions.

Return the arc tangent 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 run-time library and must be declared EXTERN before use.

**BEGOQQ**

```
PROCEDURE BEGOQQ; EXTERN;
```

A library routine (initialization).

BEGOQQ is called during initialization, and the default version does nothing. However, you can write your own version of BEGOQQ, if for example, you want to invoke a debugger or to write customized messages to the video display, such as the time of execution.

See also ENDOQQ.

**BEGXQQ**

```
PROCEDURE BEGXQQ; EXTERN;
```

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:

- resets the stack and the heap
- initializes the file system
- calls BEGOQQ
- calls the program body

Invoking this procedure to restart a program does not take care of closing any files that may have previously been opened. Similarly, it does not reinitialize variables originally set in a VALUE section or with the initialization switch on.

**BYLONG**

```
FUNCTION BYLONG(HI:WORD or INTEGER or INTEGER4;
LO:WORD or INTEGER or INTEGER4); INTEGER4
```

An **extend level** intrinsic function.

Converts WORDS or INTEGERs (or the LOWORDs of INTEGER4s) to an INTEGER4 value. BYLONG concatenates the low order words of the operand.

The low-order word of the first operand becomes the high-order word of the result. The low-order word of the second operand becomes the low-order word of the result.

If the first value is of type WORD, its most significant bit becomes the sign of the result.

To assign a WORD to an INTEGER4, use BYLONG instead of the ORD function, because ORD will sign-extend the WORD. For example,

```
Integer4Var := BYLONG (Ø, WordExpression);
```

**BYWORD**

```
FUNCTION BYWORD (PAR1, PAR2): WORD;
```

An **extend level** intrinsic function.

Converts bytes (or the LOBYTEs of INTEGERs or WORDs) to a WORD value. PAR1 and PAR2 can have 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)
```

**CHR**

```
FUNCTION CHR (X: ORDINAL): CHAR;
```

A data conversion function.

Converts any ordinal type to CHAR. The ASCII code for the result is the internal binary representation of 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) < Ø. However, the error is caught only if the range-checking switch ($RANGECK) is on.

**CHSRQQ and CHDRQQ**

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

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 run-time library and must be declared EXTERN before use.

**CLOSE**

```
PROCEDURE CLOSE (VAR F : FILE OF ..);
```

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 the subsection "Extend Level Procedures" in Section 15, "File-Oriented Procedures and Functions," for a description of CLOSE.

**CNSRQQ and CNDRQQ**

```
FUNCTION CNSRQQ (CONSTS A: REAL4): REAL4;
FUNCTION CNDRQQ (CONSTS A: REAL8): REAL8;
```

See COS.

**CONCAT**

```
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, that is, if UPPER (D) < D.LEN + UPPER (S).

**COPYLST**

```
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, that is, if UPPER (S) > UPPER (D).

**COPYSTR**

```
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, that is, if UPPER (S) > UPPER (D).

**COS**

```
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 (CONSTS REAL4) and/or CNDRQQ (CONSTS REAL8) and use them instead.

**DECODE**

```
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. When X is a subrange, DECODE returns FALSE if the value is out of range (regardless of the setting of the range-checking switch.) Leading and trailing spaces and tabs in the LSTRING are ignored. All other characters in the LSTRING must be part of the representation.

X must be one of the types INTEGER, WORD, enumerated, one of their subranges, BOOLEAN, REAL4, REAL8, INTEGER4, or a pointer. (Address types need the .R or .S suffix.)

The LSTR parameter must reside in the default data segment.

See also ENCODE.

**DELETE**

```
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, that is, if D.LEN < (I + N - 1).

**DISCARD**

```
PROCEDURE DISCARD (VAR F : FILE OF ..);
```

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

Closes and deletes an open file.

See the subsection "Extend Level Procedures," in Section 15, "File-Oriented Procedures and Functions," for a description.

**DISMQQ**

```
FUNCTION DISMQQ(block : ADSMEM); EXTERN;
```

Performs FREMQQ with error checking.

This function is from the run-time library and must be declared EXTERN before use.

**DISPOSE**

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 can not be NIL, uninitialized, or pointing at a heap item that already has been DISPOSEd. These are checked if the NIL check switch in 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 detected by our version of Pascal.

**DISPOSE**

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 those defined in the corresponding NEW procedure.

See also SIZEOF, which uses the same array upper bounds or tag value parameters to return the number of bytes in a variable.

**ENCODE**

```
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 the subsection "Read Formats" in Section 15, "File-Oriented Procedures and Functions," for a discussion of these parameters.)

X must be one of the types INTEGER, WORD, enumerated, one of their subranges, BOOLEAN, REAL4, REAL8, INTEGER4, or a pointer. (Address types need the .R or .S suffix.)

The LSTR parameter must reside in the default data segment.

See also DECODE.

**ENDOQQ**

```
PROCEDURE ENDOQQ; EXTERN;
```

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, for example, you want to invoke a debugger or to write customized messages, such as the time of execution, to the video display.

Since ENDOQQ is called after errors are processed, if ENDOQQ itself invokes an error, the result is an infinite termination loop.

See also BEGOQQ.

**ENDXQQ**

```
PROCEDURE ENDXQQ ; EXTERN;
```

The termination procedure.

ENDXQQ is the overall termination routine and performs the following actions:

- It calls ENDOQQ.
- It terminates the file system (closing any files opened by the Pascal file system).
- It returns to the 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.

**EOF**

```
FUNCTION EOF: BOOLEAN;
FUNCTION EOF (VAR F : FILE OF ..): 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 the subsection "EOF and EOLN" in Section 15, "File-Oriented Procedures and Functions," for a description.

**EOLN**

```
FUNCTION EOLN: BOOLEAN;
FUNCTION EOLN (VAR F : FILE OF ..): 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 the subsection "EOF and EOLN," in Section 15, "File-Oriented Procedures and Functions," for a description.

**EVAL**

```
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.

**EXSRQQ and EXDRQQ**

```
FUNCTION EXSRQQ (CONSTS A: REAL4): REAL4; EXTERN;
FUNCTION EXDRQQ (CONSTS A: REAL8): REAL8; EXTERN;
```

See EXP.

**EXP**

```
FUNCTION EXP (X: NUMERIC): REAL;
```

An arithmetic function.

Returns the exponential value of X (that is, 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.

**FILLC**

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

An **extend level** intrinsic procedure.

Fills D with N copies of the CHAR C.

Note that no bounds checking is done.

See also PROCEDURE FILLSC for segmented address types. The MOVE and FILL procedures (MOVESL, MOVESR, MOVEL, MOVER, FILLC AND FILLSC) 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.

**FILLSC**

```
PROCEDURE FILLSC (D: ADSMEM; N:WORD; C: CHAR);
```

An **extend level** intrinsic procedure.

Fills D with N copies of the CHAR C. No bounds checking is done.

This procedure is the same as FILLC except that the target parameter is VARS.

**FLOAT**

```
FUNCTION FLOAT (X: INTEGER): REAL;
```

A data conversion function.

Converts an INTEGER value to a REAL value. You normally do not 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.

**FLOAT4**

```
FUNCTION FLOAT4 (X: INTEGER4): REAL;
```

A data conversion function.

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

**FREECT**

```
FUNCTION FREECT (SIZE: WORD): WORD; EXTERN;
```

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 adjacent free blocks and is generally used with the SIZEOF function. However, it does not assume that any stack space will be needed. Since stack space generally is needed, the value returned should be reduced accordingly.

Example:

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

**FREMQQ**

```
FUNCTION FREMQQ(block : ADSMEM) : WORD; EXTERN;
```

Frees a memory block from the long heap. Returns Ø if no errors encountered, nonzero otherwise.

This function is from the run-time library and must be declared EXTERN before use.

**GET**

```
PROCEDURE GET (VAR F : FILE OF ..);
```

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 the subsection "GET and PUT," in Section 15, "File-Oriented Procedures and Functions," for a description.

**GETMQQ**

```
FUNCTION GETMQQ (wants : WORD) : ADSMEM; EXTERN;
```

Performs ALLMQQ with error checking.

This function is from the run-time library and must be declared EXTERN before use.

**GTYUQQ**

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

A library function (terminal I/O).

Reads a maximum of LEN characters from the 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 Pascal file system.

**HIBYTE**

```
FUNCTION HIBYTE (I : INTEGER or WORD): BYTE;
```

An **extend level** intrinsic function.

Returns the most significant byte of an INTEGER or WORD.

See also LOBYTE.

**HIWORD**

```
FUNCTION HIWORD (I : INTEGER4): WORD;
```

An **extend level** intrinsic 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 LOWORD.

**INSERT**

```
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, that is, if

```
UPPER (D) < UPPER (S) + D.LEN
```

or if

```
D.LEN + 1 < I
```

**LADDOK**

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

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, modulo 2**32 arithmetic, or arithmetic based on user input data.

**LDSRQQ and LDDRQQ**

```
FUNCTION LDSRQQ (CONSTS A: REAL4): REAL4; EXTERN;
FUNCTION LDDRQQ (CONSTS A: REAL8): REAL8; EXTERN;
```

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 must be declared EXTERN before use.

**LMULOK**

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

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 on overflow. Normal arithmetic can 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, modulo 2**32 arithmetic, or arithmetic based on user input data.

**LN**

```
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.

**LNSRQQ and LNDRQQ**

```
FUNCTION LNSRQQ (CONSTS A: REAL4): REAL4; EXTERN;
FUNCTION LNDRQQ (CONSTS A: REAL8): REAL8; EXTERN;
```

See LN.

**LOBYTE**

```
FUNCTION LOBYTE (I : INTEGER or WORD): BYTE;
```

An **extend level** intrinsic function.

Returns the least significant byte of an INTEGER or WORD.

See also HIBYTE.

**LOCKED**

```
FUNCTION LOCKED (VARS SEMAPHORE: WORD): BOOLEAN;
EXTERN;
```

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 it is a binary semaphore, there are only two states.

See also UNLOCK.

**LOWER**

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:

- the lower bound of an array
- the first allowable element of a set
- the first value of an enumerated type
- the lower bound of a subrange

LOWER uses the type and not the value of the expression. The value returned by LOWER is always a constant.

See also UPPER.

**LOWORD**

FUNCTION LOWORD (I : INTEGER4): WORD;

An **extend level** intrinsic function.

Returns the low-order WORD of the four bytes of the INTEGER4.

See also HIWORD.

**MARKAS**

```
PROCEDURE MARKAS (VAR HEAPMARK: INTEGER4); EXTERN;
```

A library procedure (heap management).

Parallels the MARK procedure in other Pascals. MARKAS marks the upper and lower limits of the short heap. The DISPOSE procedure is generally more powerful, but MARKAS can be useful for converting from other Pascal dialects.

In other Pascals, the parameter is of a pointer type. However, in our version of Pascal, two words are needed to save the heap limits, since 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, M for example, as a VAR parameter to MARKAS. MARKAS places the bounds of the heap in M. The low-order word of M receives the relative address of the beginning of the heap, the high-order word of M that of the end of the heap. To release heap space, simply invoke the procedure with RELEAS (M).

Note that to use MARKAS and RELEAS you should not use the long heap at all. These two procedures also work as intended only if you never call DISPOSE.

**MDSRQQ and MDDRQQ**

```
FUNCTION MDSRQQ (CONSTS A, B: REAL4): REAL4;
EXTERN;

FUNCTION MDDRQQ (CONSTS A, B: REAL8): REAL8;
EXTERN;
```

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 run-time library and must be declared EXTERN before use.

**MEMAVL**

```
FUNCTION MEMAVL: WORD; EXTERN;
```

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 can return a value less than the actual number of bytes available.

**MNSRQQ and MNDRQQ**

```
FUNCTION MNSRQQ (CONSTS A, B: REAL4): REAL4;
EXTERN;

FUNCTION MNDRQQ (CONSTS A, B: REAL8): REAL8;
EXTERN;
```

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 run-time library and must be declared EXTERN before use.

See also MXSRQQ and MXDRQQ.

**MOVEL**

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

An **extend 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, 156)
```

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 (MOVESL, MOVESR, MOVEL, MOVER, FILLC, AND FILLSC) take value parameters of type ADRMEM and ADSMEM. However, 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.

**MOVER**

```
PROCEDURE MOVER (S, D: ADRMEM; N: WORD);
```

An **extend level** intrinsic procedure.

Like MOVEL, but starts at the highest addressed byte of each array. Use MOVER and MOVESR to shift bytes right or when the address ranges do not overlap. 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 MOVE and FILL procedures (MOVESL, MOVESR, MOVEL, MOVER, FILLC, AND FILLSC) take value parameters of type ADRMEM and ADSMEM. However, 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.

**MOVESL**

PROCEDURE MOVESL (S, D: ADSMEM; :N: WORD);

An **extend 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)
```

This procedure is the same as MOVEL, except that the target parameter is VARS.

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 (MOVESL, MOVESR, MOVEL, MOVER, FILLC, AND FILLSC) take value parameters of type ADRMEM and ADSMEM. However, 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.

**MOVESR**

```
PROCEDURE MOVESR (S, D: ADSMEM; N: WORD);
```

An **extend level** intrinsic procedure.

Like MOVESL, but starts at the highest addressed byte of each array. Use MOVER and MOVESR to shift bytes right or when the address ranges do not overlap.

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.

This procedure is the same as MOVER, except that the target parameter is VARS.

The MOVE and FILL procedures (MOVESL, MOVESR, MOVEL, MOVER, FILLC, AND FILLSC) take value parameters of type ADRMEM and ADSMEM. However, 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.

**MXSRQQ and MXDRQQ**

```
FUNCTION MXSRQQ (CONSTS A, B: REAL4): REAL4;
EXTERN;

FUNCTION MXDRQQ (CONSTS A, B: REAL8): REAL8;
EXTERN;
```

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 run-time library and must be declared EXTERN before use.

See also MNSRQQ and MNDRQQ.

**NEW**

```
PROCEDURE NEW (VARS P: POINTER);
```

A library procedure (heap management, short form).

Allocates a new variable (for example, 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, T1 .. 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, 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:

- It assumes that the first nonconstant tag field and all following tags have unknown values, and allocates the maximum size necessary.
- 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:

- o assign tag values
- o check that they are initialized correctly
- o 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

- o used as an expression operand
- o passed as a parameter
- o assigned a value

The compiler does not catch these errors. Fields within the record can be used normally.

Assigning a larger record to a smaller one allocated with the long form of NEW would wipe out part of the heap. This condition is difficult to detect at compile time. Therefore, any assignment to a record in the heap that has variants uses the actual length of the record in the heap, rather than the maximum length.

However, an assignment to a field in an invalid variant can 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 referenced parameter if the formal parameter is of the same super array type. Components of the array can be used normally.

**ODD**

```
FUNCTION ODD (X: ORDINAL): BOOLEAN;
```

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.

This function can also be used with INTEGER4.

**ORD**

```
FUNCTION ORD (X: VALUE): INTEGER;
```

A data conversion function.

Converts to INTEGER any value of one of the types shown in Table 14-5, according to the rules given.

**Table 14-5. Conversion to INTEGER.**

| Type of X | Return value |
|---|---|
| INTEGER | X |
| WORD <= MAXINT | X |
| WORD > MAXINT | X - 2 * (MAXINT + 1) (that is, 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 (that is, same as ORD(LOWORD(INTEGER4)) |
| Pointer | Integer value of pointer |

**PACK**

```
PROCEDURE PACK
(CONSTS A: UNPACKED; I: INDEX; VARS Z: PACKED);
```

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; that is, 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.

**PAGE**

```
PROCEDURE PAGE;
PROCEDURE PAGE (VAR F : FILE OF ..);
```

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 (OUTPUT).

See subsection "PAGE," in Section 15, "File-Oriented Procedures and Functions," for a description of PAGE.

**PISRQQ and PIDRQQ**

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

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

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 run-time library and must be declared EXTERN before use.

**PLYUQQ**

```
PROCEDURE PLYUQQ; EXTERN;
```

A library routine (terminal I/O).

Write an end-of-line character to the video display.

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 Pascal file system.

**POSITN**

```
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.

**PREALLOCHEAP**

```
FUNCTION PREALLOCHEAP (VARS CBALLOC: WORD): WORD;
EXTERN;
```

A library function.

Allocates short-lived memory from the operating system memory pool. This memory is unused after this call. It then can be used for the heap by heap management routines.

This preallocation is useful if your program then calls the operating system directly to allocate short-lived memory. (See Section 4, "Memory Management" in the CTOS Operating System Manual for further information on memory organization and management.)

Lets you specify how much storage is to be allocated for the short heap. You can then use short-lived memory without worrying about running out of heap space.

CBALLOC — Is the count of bytes to allocate for the heap

If cbAlloc is #ØFFFF, the maximum possible storage is allocated for the heap

**PREALLOCLONGHEAP**

```
FUNCTION PREALLOCLONGHEAP (CPARA: WORD) : WORD;
EXTERN;
```

A run-time library function.

Normally, the first call to a long heap allocation routine allocates as much short-lived memory as possible for the short heap and takes all the rest of the short-lived memory for the long heap (to satisfy the current and possible future requests). To avoid the rest of the short-lived memory being allocated for the long heap, you can preallocate the short-lived memory for the long heap using PREALLOCLONGHEAP.

CPARA is the number of paragraphs (number of bytes divided by 16) to be allocated for the long heap. This procedure

- allocates as much short-lived memory as possible for the short heap
- allocates CPARA paragraphs of short-lived memory for the long heap

  If there are less than CPARA paragraphs available, all available short-lived memory is allocated.

  If CPARA = #ØFFFF, then all available short-lived memory is allocated.

**PRED**

```
FUNCTION PRED (X: ORDINAL): ORDINAL;
```

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.

This function can also be used with INTEGER4.

**PRSRQQ and PRDRQQ**

```
FUNCTION PRSRQQ (A, B: REAL4): REAL4; EXTERN;
FUNCTION PRDRQQ (A, B: REAL8): REAL8; EXTERN;
```

Arithmetic functions.

The return value is A**B (A to the REAL power of B). Both A and B are of type REAL4 or REAL7, as shown. An error occurs if A < Ø (even if B happens to have an integer value).

These functions are from the run-time library and must be declared EXTERN before use.

**PTYUQQ**

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

A library routine (terminal I/O).

Writes LEN characters, beginning at LOC in memory, to the video display.

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 Pascal file system.

**PUT**

```
PROCEDURE PUT (VAR F : FILE OF ..);
```

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 the subsection "GET and PUT" in Section 15, "File-Oriented Procedures and Functions," for a description.

**READ**

```
PROCEDURE READ (VAR F : FILE OF ..; P1, P2, ..
PN);
```

A file system procedure.

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

See the subsection "Textfile Input and Output" in Section 15, "File-Oriented Procedures and Functions," for a description.

**READFN**

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

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

READFN is the same as READ (not READLN) with two exceptions:

- File parameter F should be present (INPUT is assumed but a warning is given.)
- 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 in the READ procedure.

See the subsection "Extend Level Procedures" in Section 15, "File-Oriented Procedures and Functions," for a description.

**READLN**

```
PROCEDURE READLN (VAR F : FILE OF ..; P1, P2 ..
PN);
```

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 the subsection "Textfile Input and Output," in Section 15, "File-Oriented Procedures and Functions," for a description.

**READSET**

```
PROCEDURE READSET
(VAR F : FILE OF ..; 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 the subsection "Extend Level Procedures" in Section 15, "File-Oriented Procedures and Functions," for a description.

**RELEAS**

```
PROCEDURE RELEAS (VAR HEAPMARK: INTEGER4); EXTERN;
```

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 this version of Pascal is generally more powerful, but RELEAS can be useful for converting from other Pascal dialects.

In other Pascals, the parameter is of a pointer type. However, this version needs two words to save the heap limits, since 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, M for example, 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 DISPOSE is never called.

**RESET**

```
PROCEDURE RESET (VAR F : FILE OF ..);
```

A file system procedure.

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

See the subsection "RESET and REWRITE" in Section 15, "File-Oriented Procedures and Functions," for a description of RESET.

**RESULT**

```
FUNCTION RESULT (FUNCTION-IDENTIFIER): VALUE;
```

An **extend level** intrinsic function.

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

**RETYPE**

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

An **extend 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).

> NOTE
>
> RETYPE is a "dangerous" type escape and may not work as intended.

Example:

```
CONST MODEREAD = 'mr';

TYPE COLOR = (RED, BLUE, GREEN);
       S2 = STRING (2);
VAR C : CHAR;
    I, J : INTEGER;
    R : REAL4;
    TINT: COLOR;
    W : WORD;
    .
    .
    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.}

W := RETYPE (WORD, MODEREAD);
{W receives the value #6D72 since}
{this is the ASCII representation}
{of 'mr"}
```

There are two other ways to change type in this version of Pascal.

- You can declare an address variable of the type wanted and assign to it the address of any other variable (using ADR).

- 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.)

Each of these methods has its own subtle differences and quirks and should be avoided whenever possible.

**REWRITE**

```
PROCEDURE REWRITE (F);
```

A file system procedure.

Resets the current file position to its beginning.

See the subsection "RESET and REWRITE" in Section 15, "File-Oriented Procedures and Functions," for a description of REWRITE.

**ROUND**

```
FUNCTION ROUND (X: REAL): INTEGER;
```

An arithmetic function.

Rounds X to the nearest integer. X is of type REAL4 or REAL8; the return value is of type INTEGER. The numbers with a fractional part of Ø.5 are rounded to the nearest even integer.

Examples:

```
ROUND (1.6) is 2
ROUND (-1.6) is -2
ROUND (1.5) = ROUND (2.5) = 2
```

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

**ROUND4**

```
FUNCTION ROUND4 (X: REAL): INTEGER4;
```

An arithmetic function.

Rounds real X to the nearest integer. X is of type REAL4 or REAL8; the return value is of type INTEGER4. Numbers with a fractional part of Ø.5 are rounded to the nearest even integer.

Examples:

```
ROUND4 (1.6) is 2
ROUND4 (-1.6) is -2
```

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

**SADDOK**

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

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 can 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.

**SCANEQ**

```
FUNCTION SCANEQ
(LEN: INTEGER; PART: CHAR; CONSTS S: STRING; I:
INTEGER): INTEGER;
```

A string intrinsic function.

Scans S, 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 or if I > UPPER (S). There are no error conditions.

**SCANNE**

```
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 or if I > UPPER (S). There are no error conditions.

**SEEK**

```
PROCEDURE SEEK (VAR F : FILE OF ..; 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 the subsection "Extend Level I/O" in Section 15, "File-Oriented Procedures and Functions," for details.

**SHSRQQ and SHDRQQ**

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

Arithmetic functions.

Return the hyperbolic sine of A. A is of type REAL4 or REAL8, as shown.

These functions are from the run-time library and must be declared EXTERN before use.

**SIN**

```
FUNCTION SIN (X: NUMERIC): REAL;
```

An arithmetic function.

Returns the sine of X. X is 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.

**SIZEOF**

```
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.

**SMULOK**

```
FUNCTION SMULOK
(A, B: INTEGER; VAR C: INTEGER): BOOLEAN; EXTERN;
```

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 can cause a run-time error, even if the arithmetic debugging switch is off. It returns TRUE if there is no overflow, and FALSE if there is.

This routine can be useful for extended-precision arithmetic, or modulo 2**16 arithmetic, or arithmetic based on user input data.

**SNSRQQ and SNDRQQ**

```
FUNCTION SNSRQQ (CONSTS A: REAL4): REAL4; EXTERN;
FUNCTION SNDRQQ (CONSTS A: REAL8): REAL8; EXTERN;
```

See SIN.

**SQR**

```
FUNCTION SQR (X: NUMERIC): NUMERIC;
```

An arithmetic function.

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

**SQRT**

```
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 Ø.

**SRSRQQ and SRDRQQ**

```
FUNCTION SRSRQQ (CONSTS A: REAL4): REAL4; EXTERN;
FUNCTION SRDRQQ (CONSTS A: REAL8): REAL8; EXTERN;
```

See SQRT.

**SUCC**

```
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.

This function can also be used with INTEGER4.

**THSRQQ and THDRQQ**

```
FUNCTION THSRQQ (CONSTS A: REAL4): REAL4; EXTERN;
FUNCTION THDRQQ (CONSTS A: REAL8): REAL8; EXTERN;
```

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 run-time library and must be declared EXTERN before use.

**TNSRQQ and TNDRQQ**

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

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 run-time library and must be declared EXTERN before use.

**TRUNC**

```
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.

**TRUNC4**

```
FUNCTION TRUNC4 (X: REAL): INTEGER4;
```

An arithmetic function.

Truncates real X toward 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.

**UADDOK**

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

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 can 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, modulo 2**16 arithmetic, or arithmetic based on user input data.

**UMULOK**

```
FUNCTION UMULOK (A, B: WORD; VAR C: WORD):
BOOLEAN; EXTERN;
```

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 can cause a run-time error even if the arithmetic debugging switch is off. This routine returns TRUE if there is no overflow and FALSE if there is.

UMULOK is useful for extended-precision arithmetic, modulo 2**16 arithmetic, or arithmetic based on user input data.

**UNLOCK**

```
PROCEDURE UNLOCK (VARS SEMAPHORE: WORD); EXTERN;
```

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 LOCKED.

**UNPACK**

```
PROCEDURE UNPACK
(CONSTS Z: PACKED; VARS A: UNPACKED; I: INDEX);
```

A data conversion procedure.

Moves elements from a 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; that is, 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 PACK.

**UPPER**

```
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:

- the upper bound of an array
- the last allowable element of a set
- the last value of an enumerated type
- 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. UPPER uses the type and not the value of the expression.

See also LOWER.

**WRD**

```
FUNCTION WRD (X: VALUE): WORD;
```

A data conversion procedure.

Converts to WORD any of the types shown in Table 14-6, according to the rules given.

**Table 14-6. Conversion to WORD.**

| Type of X | Return value |
|---|---|
| WORD | X |
| INTEGER >= Ø | X |
| INTEGER < Ø | X + MAXWORD + 1 (that is, the 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 (that is, the same as LOWORD(INTEGER4) |
| Pointer | Word value of pointer |

**WRITE and WRITELN**

```
PROCEDURE WRITE (VAR F : FILE OF ..; P1, P2, ..
PN);
PROCEDURE WRITELN (VAR F : FILE OF ..; P1, P2, ..
PN);
```

File system level intrinsic procedures.

Write 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.

The first parameter to WRITE specifies the output file to be written. This parameter can be omitted only if the first parameter to be written is not of the type FILE. If the output file parameter is omitted, the default output file is OUTPUT.

Parameters to WRITE that are address types should be written using the .S and .R notation.

See the subsection "WRITE and WRITELN" in Section 15, "File-Oriented Procedures and Functions," for a description.

# 15 FILE-ORIENTED PROCEDURES AND FUNCTIONS

This section discusses the file I/O procedures and functions, as well as two special features our version of Pascal provides that facilitate your use of files, lazy evaluation, and concurrent I/O. (All procedures and functions that are available to you either because they are predeclared or because they are part of the run-time library, except those that relate to file input and output, are discussed in detail in Section 14, "Introduction to Procedures and Functions.")

The Pascal file system supports a variety of procedures and functions that operate on files of different modes and structures. These procedures and functions can be categorized as 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 | |

## 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. Lazy evaluation is also discussed in this section. In all the descriptions that follow, F is a file parameter (files are always reference parameters), and F^ is the buffer variable.

All file variables operated on by these procedures must reside in the default data segment. This restriction increases the efficiency of file system calls.

GET and PUT

Read 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.

RESET and REWRITE

Set 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 EOLN

Used to check for end-of-file and end-of-line conditions. They return a BOOLEAN result. In general these values indicate when to stop reading a line or a file.

PAGE

Helps in formatting textfiles. It is not a necessary procedure in the same sense as GET and PUT.

**GET AND PUT**

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

- The current file position is advanced to the next component.
- The value of this component is assigned to the buffer variable F^.
- EOF (F) becomes FALSE.

Advancing and assigning may be deferred internally, depending on the mode of the file. (See the subsection "Lazy Evaluation" below.)

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, since DIRECT mode permits repeated GET operations at the end of the file, if F has mode DIRECT, EOF (F) can be TRUE or FALSE. 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. The following rules apply:

- 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.

- For DIRECT mode files, PUT can occur immediately after a RESET or GET.

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

EOF (F) must be TRUE before PUT (F), unless F is a DIRECT mode file. EOF (F) is always TRUE after PUT (F). If F^ is a record with variants, the variant with the maximum size is written.

**RESET AND REWRITE**

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

PROCEDURE RESET (VAR F);

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 before it is read. For DIRECT files, writing can be done after RESET as well.

A RESET closes the file and then opens it. If the file did not exist it is created and then opened. An error occurs if the file name 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 also be done after REWRITE.)

A REWRITE closes the file and then opens it. If the file does not exist, it is created. If it does exist, its old value is lost (unless it has mode DIRECT). The file name 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 that file can 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.

**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.

For SEQUENTIAL and TERMINAL file modes, returns TRUE if the buffer variable F^ is positioned at the end of the file F, FALSE otherwise. 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 if INPUT is reassigned to another file, or when FINISH (Ø4) AND CANCEL (Ø7) are encountered. Calling the EOF (F) function accesses the buffer variable F^.

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

A primitive file system intrinsic function.

Returns TRUE if the current position of the file is at the end of a line in the textfile F after a GET (F), FALSE otherwise. The file must have ASCII structure.

According to the ISO standard, calling EOLN (F) when EOF (F) is TRUE is an error. The compiler detects these errors 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^.

**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:TEXT);
```

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. PAGE (F) writes a formfeed, CHR (12).

**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 does not work for interactive I/O.

Lazy evaluation is used by our version of Pascal as a way of 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 the keyboard, this first component does not yet exist.

Therefore, at a terminal or workstation, you must first type a character to accommodate the GET procedure. Only then will you be prompted for any input.

Lazy evaluation eliminates this problem for text-files 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.

**Table 15-2. Lazy Evaluation.**

| Statement | Status At Call | Action | Status On Exit |
|---|---|---|---|
| GET (F) | Full | Point to next file component. Becomes EMPTY, since value pointed to is not in buffer variable. | Empty |
| GET (F) | Empty | Load buffer variable with current file component, then point to next file component. Becomes EMPTY, since value pointed to is not in buffer variable. | Empty |
| Reference to F^ | Full | No action required. | Full |
| Reference to F^ | Empty | Load buffer variable with current file component. | Full |

Note that RESET (F) first sets the status full and then calls GET, which sets the status to empty without any physical input.

Example of lazy evaluation with automatic RESET call:

```
{INPUT is automatically a textfile.}
{RESET (INPUT); done automatically.}
WRITE (OUTPUT, "Enter number: ");
READLN (INPUT, FOO);
```

The automatic initial call to the RESET procedure calls a GET procedure, which changes the buffer variable status from full to empty. The first physical action to the terminal is the prompt output from the WRITE. READLN does a series of the following operations:

```
temp := INPUT^;
GET (INPUT)
```

Physical input occurs when each INPUT^ is fetched and the GET procedure sets the status back to empty.

READLN ends with the sequence:

```
WHILE NOT EOLN DO GET (INPUT);
GET (INPUT)
```

This operation skips trailing characters and the RETURN (ØAh). 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.

## 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

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

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 than GET and PUT in the syntax for their parameter lists, allowing, among other things, for a variable number of parameters. Moreover, the parameters need not necessarily be of type CHAR, but can also be of certain other types, in which case the data transfer is accompanied by an implicit data conversion operation. In some cases, parameters can include additional formatting values that affect the data conversions used.

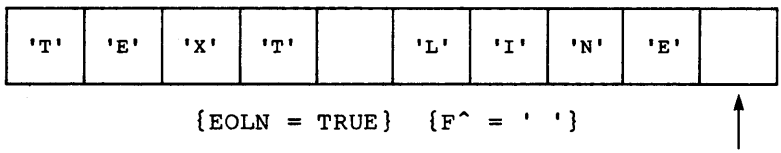
If the first variable is a file variable, then it is the file to be read or written. Otherwise, the standard file INPUT is automatically assumed as the default value for reading, and OUTPUT the default value for writing.

These two files have TERMINAL mode and ASCII structure and are predeclared as:

```
VAR INPUT, OUTPUT:  TEXT;
```

The compiler treats INPUT and OUTPUT the same as 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 file name. Instead, they are assigned to the keyboard and video, respectively. 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 (that is, past the last character of a line), then the value of the buffer variable F^ becomes a blank, and the standard function EOLN (F) yields the value true. For example:




Advancing the file position once more causes one of three things to happen:

- If the end of the file is reached, then EOF (F) becomes TRUE.
- If the next line is empty, that is, if the next character in the file is the RETURN (ØAh) a blank is assigned to F^, and EOLN (F) remains TRUE.
- Otherwise, the first character of the next line is assigned to F^ and EOLN (F) is set to FALSE.

Since line markers are not elements of type CHAR in standard Pascal, they can, in theory, only be generated by the procedure WRITELN. However, in this version of Pascal, the actual character RETURN (ØAh) is used for the line marker. You can, therefore. WRITE a line marker, for example, WRITE(f,CHR(1Ø)); but not READ one.

When a textfile being written is closed, a final line marker is automatically appended to the last line of any nonempty file in which the last character is not already a line marker.

When you reach the end of a textfile during a READ, a line marker for the last line is returned even if one was not present in the file. Lines in a textfile always end with a line marker.

Any list of data written by a WRITELN is usually readable with the same list in a READLN (unless an LSTRING occurs that is not on the end of the list.)

Interactive prompt and response is very easy in Pascal. To have input on the same line as the prompt, use WRITE for the prompt. READLN must always be used for the response. For example:

```
WRITE ('Enter command:  ');
READLN (response);
```

If no file is given, most of the textfile procedures and functions assume either the INPUT file or the OUTPUT file. For example, if I is of type INTEGER, then READ (I) is the same as READ (INPUT, I).

**READ AND READLN**

PROCEDURE READ
PROCEDURE READLN

File system intrinsic procedures for textfile I/O.

READ and READLN read data from textfiles. Both are defined in terms of the more primitive operation, GET. That is, if P is of type CHAR, then READ (F, P) is equivalent to:

```
BEGIN
  P := F^;
  {Assign buffer variable F^ to P.}
  GET (F)
  {Assign next component of file to F^.}
END
```

READ can take more than one 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 analogous to READ, except that it reads up to and including the end of line. At the primitive GET level, without parameters, READLN is equivalent to the following:

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

A READLN with parameters, as in READLN (F, P1, P2, .. Pn), is equivalent to the following

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

READLN is often used to skip to the beginning of the next line. It can only be used with textfiles (ASCII mode).

If no other file is specified, both READ and READLN read from the standard INPUT file. Therefore, the name INPUT need not be designated explicitly. For example, these two READ statements perform identical actions:

```
READ (P1, P2, P3)
{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
REAL8
```

When the compiler reads a variable of a sub-range type, the value read must be in range. Otherwise, an error occurs, regardless of the setting of the range-checking switch ($RANGECK).

The procedure READ can also read from a file that is not a textfile (for example, a BINARY file). The form READ (F, P1, P2, .. Pn) can be used on a BINARY file. However, this READ does not work as expected after a SEEK on a DIRECT mode file.

For BINARY files, READ (F, X) is equivalent to:

```
BEGIN
  X := F^;
  GET (F)
END
```

**READ FORMATS**

The READ process for formatted types (everything except CHAR, STRING, and LSTRING) first reads characters into an internal LSTRING and then decodes the string to get the value. (See also the discussion of DECODE in Section 14, "Available Procedures and Functions.")

Two important points apply to formatted reads:

- 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.

- 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.

Most of the formatting rules below apply to the function DECODE, as well as to READ and READLN.

INTEGER and WORD types

If P is of type INTEGER, WORD, or a subrange thereof, then READ (F, P) implies reading a sequence of characters from F, which form a number according to the normal Pascal syntax, and then assigning the number to P. Nondecimal notation (16#C007, 8#74, 10#19, 2#101, #Face) is accepted for both INTEGER

and WORD, with a radix of 2 through 36. If P is of an INTEGER type, a leading plus (+) or minus (-) sign is accepted. If P is of a WORD type, numbers up to MAXWORD are accepted (Ø..65535).

REAL and INTEGER4 types

If P is of type REAL, or at the **extend level** of type REAL or INTEGER4, READ (F, P) implies reading a sequence of characters from F that form a number of the appropriate type and assigning the number to P. Nondecimal notation is not accepted for REAL numbers, but is accepted for INTEGER4 numbers. When reading a REAL value, a number with a leading or trailing decimal point is accepted, even though this form gives a warning if used as a constant in a program.

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 is 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.

Reference types

At the **extend level**, if P is a pointer type, a number is read as a WORD and assigned to P, so that writing a pointer and later reading it yields the same pointer value. The address types should be read as WORDs using .R or .S notation.

String types

At the **extend level**, if P is a STRING (n), the next "n" characters are read sequentially into P. Preceding line markers, spaces, tabs, or formfeeds 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 256 characters on the length of a STRING read. P can be the super array type STRING (for example, a reference parameter or pointer referent variable).

At the **extend level**, if P is an LSTRING (n), 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 have been read, the length of the LSTRING is set to the number of characters read and the file position remains at the line marker.

If the LSTRING is filled with n characters before the line marker is encountered, the file position remains at the next character. P can be the super array type LSTRING (for example, a reference parameter or pointer referent variable). READ (LSTRING) is handy when reading entire lines from a text-file, especially when the length of the line is needed. For example, the easiest way to copy a textfile is by using READLN and WRITELN with an LSTRING variable.

**WRITE AND WRITELN**

PROCEDURE WRITE
PROCEDURE WRITELN

File system intrinsic procedures for textfile I/O.

WRITE and WRITELN write data to textfiles. Both are defined in terms of the more primitive operation, PUT. That is, if P is of type CHAR and F is 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 line. In all other respects, WRITELN is analogous to WRITE. Thus, WRITELN (F, P1, P2, .. Pn) is equivalent to:

```
BEGIN
  WRITE (P1, P2, ..Pn);
  WRITELN (F)
END
```

If either WRITE or WRITELN has no file parameter, the default file parameter is OUTPUT. Therefore, the first statement in each of the following pairs is equivalent to the second:

```
WRITE (P1, P2, .. Pn);
WRITE (OUTPUT, P1, P2, .. Pn);

WRITELN (P1, P2, .. Pn);
WRITELN (OUTPUT, P1, P2, .. Pn)
```

At the standard level, parameters in a WRITE can be expressions of any of the following types:

| | |
|---|---|
| CHAR | BOOLEAN |
| INTEGER | STRING |
| REAL | |

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

WORD
an enumerated type
INTEGER4
a pointer type
LSTRING
REAL8

Parameters may take optional M and N values (see the subsection "WRITE Formats," for information about M and N parameters).

Although the procedure WRITE can also write to a BINARY file (that is, not a textfile), this is not recommended for DIRECT files after a SEEK operation, because the complementary READ form does not work as you might expect.

For BINARY files, WRITE (F, X) is equivalent to:

```
BEGIN
  F := X;
  PUT (F)
END
```

The form WRITE (F, P1, P2, .. Pn) is also acceptable. Normally, BINARY writes do not accept M and N values.

**WRITE FORMATS**

In textfiles, data parameters to WRITE and WRITELN may take one of the following forms:

```
P     P:M     P:M:N     P::N
```

P can be numeric (INTEGER, REAL, REAL8, subrange), CHAR, BOOLEAN, or STRING. Alternatively, P can be an enumerated type, WORD, a pointer, or an LSTRING. M and N are expressions whose integer values are field-width parameters.

M and N values are value parameters of type INTEGER and are used for formatting in various ways.

In WRITE, the M value is the field width used as the number of characters to write.

The N value signifies:

- the number of decimal places if P is of type REAL
- the output radix if P is of type INTEGER, WORD, INTEGER4, or pointer

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 detected 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 detected by this compiler.

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.

Most of the following formatting rules apply to the function ENCODE as well as to WRITE.

INTEGER and WORD types

If P is of type INTEGER, WORD, or a subrange thereof, then the decimal representation of P is written on the file. If P is a negative INTEGER, a leading minus sign is always written. WORD values are never negative. For INTEGER and WORD values, the default M value is 8.

If ABS (M) is smaller than the representation of the number, additional character positions are used as needed. N is used to write in hexadecimal, decimal, octal, binary, or other base numbering using N equal to a number from 2 to 36; this is an extension to the ISO standard. If N is not 10 (or omitted or MAXINT), then padding on the left is with zeros and not spaces. Omitting N or setting N to MAXINT or 10 implies a decimal radix.

WORD decimal numbers from 32768 to 65535 are written normally and not in their negative integer equivalents. All values written should be separated by spaces or some other character not valid in numbers, so that values are read as separate numbers.

REAL and INTEGER4 types

If P is of type REAL or REAL8, 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.

Some examples of WRITE operations on REAL and REAL8 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.

Enumerated and Boolean types

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.

Reference types

At the **extend level**, if P is a pointer type, then it is written as a WORD. 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.

String types

If P is of type STRING (n), 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, only the first ABS (M) characters are written. If M is zero, nothing is written. If the STRING is truncated, it is always truncated on the right, even if M is negative.

At the **extend level**, if P is of type LSTRING (n), 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, only the first ABS (M) characters are written. If M is zero, nothing is written. If the STRING is truncated, it is always truncated on the right, even if M is negative. If ABS (M) is greater than the current length, spaces, not characters, fill the remaining positions past the length in the LSTRING. Note that a string of M blanks can be written with NULL:M.

## EXTEND LEVEL I/O

At the **extend level**, these additional I/O features are available:

- You can access three FCB (File Control Block) fields: F.MODE, F.TRAP, and F.ERRS.
- A number of additional procedures are predeclared.
- Temporary files are available.

The subsection "Extend Level I/O" in Section 7, "Files," discusses FCB fields in the context of files. The additional procedures and temporary files are described in below.

### EXTEND LEVEL PROCEDURES

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

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

Assigns an operating system file name in a STRING (or LSTRING) to a file F. As a rule, ASSIGN truncates any trailing blanks. ASSIGN overrides any file name set previously. A file name 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 (FCB), they are closed automatically when a RETURN or DISPOSE releases stack or heap file variables.

File variables with the STATIC attribute in procedures and functions are also closed automatically when the procedure or function returns. Files allocated statically at the program, module, or implementation level are automatically closed when the entire program terminates.

If necessary, when a CLOSE is executed, a file being written to has its operating system buffers flushed. However, the buffer variable is not PUT. If a file of type TEXT is being written and the last nonempty line does not end with a line marker, one is added to the end of the last line. If the file has the mode SEQUENTIAL and is being written, an end-of-file is written.

Note that some run-time errors may remove control from the 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:

- File parameter F should be present (INPUT is assumed, but a warning is given if F is omitted).
- If a parameter P is of type FILE, a sequence of characters forming a valid file name 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 file name and assigning the file name to a 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, formfeeds, 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 file names with READFN. It is handy when reading and parsing input lines for simple command scanners.

The L and S parameters must reside in the default data segment.

PROCEDURE SEEK (VAR F; N: INTEGER4);

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:

| 'A' | 'B' | 'C' | 'D' | 'E' | 'F' | 'G' | |
|---|---|---|---|---|---|---|---|
| n = 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |

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 now equal to 'A'; C does not equal 'E'}
```

Note that the fifth component is not assigned to C, as you might expect. To obtain this value, the following sequences of commands should be executed:

```
RESET (F);
{Initial SEEK 1, followed by GET;}
{F^ now holds 'A'.}
SEEK (F, 5);
{File positioned at 5.}
GET (F);
{File buffer variable is loaded with "E".}
C := F^
{C gets value 'E'.}
```

The rule is to always follow a SEEK (F, N) with a GET to ensure that the nth component is contained in the buffer variable.

GET and PUT operate normally on DIRECT mode files with BINARY structured files. However, READ and WRITE work only with ASCII files, that is, textfiles. READ, in particular, will not work with DIRECT mode BINARY files, because it assigns the buffer variable's value before it performs a GET. On the other hand, GET and PUT are not normally used with ASCII structured DIRECT mode files. Lazy evaluation makes READ and WRITE more appropriate. Care should always be taken when mixing normal sequential operations with DIRECT mode SEEK operations.

**TEMPORARY FILES**

Sometimes a program needs a "scratch" file for temporary, intermediate data. If this is the case, you 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 a zero has been assigned as its 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

The 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. Note that at the standard level, you can compile only entire programs; however, at the **extend level** you can also compile modules and units.

Example of a compilable program:

```
PROGRAM MAIN (INPUT, OUTPUT);
BEGIN
  WRITELN('Main Program')
END. {Main}
```

Example of a compilable module:

```
MODULE MOD_DEMO;
{No parameter list in heading}
  PROCEDURE MOD_PROC;
  BEGIN
    WRITELN
      ('Output from MOD_PROC in MOD_DEMO.')
  END;
END. {Mod_Demo}
```

Example of a compilable unit:

```
INTERFACE;
  UNIT UNIT_DEMO (UNIT_PROC);
  {UNIT_PROC is the only exported identifier}
  PROCEDURE UNIT_PROC;
END;
IMPLEMENTATION OF UNIT_DEMO;
  PROCEDURE UNIT_PROC;
  BEGIN
    WRITELN
      ('Output from UNIT_PROC in UNIT_DEMO.')
  END;
END. {Unit_Demo}
```

If you compile the last two examples shown above (MODULE MOD_DEMO and UNIT UNIT_DEMO) separately, you can later incorporate them into the main program shown below by linking all three compilands:

```
{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 executed, the output consists of the following:

```
Output from Main Program

Output from MOD_PROC in MOD_DEMO

Output from UNIT_PROC in UNIT_DEMO
```

This section describes the rules governing the construction and use of programs, modules, and units. There are programming examples for Pascal in Appendix H, "Programming Examples."

## 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 first BEGIN and last 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. (However, this is not recommended.) You can also 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 file name to the file. However, at the **extend level**, you can use the ASSIGN and READFN

procedures to assign file names, so file variables need not appear as program parameters.

The program parameters denote entities outside the program through which the program communicates with its environment. 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, except INPUT and OUTPUT, 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. (See Section 7, "Files," for a discussion of INPUT and OUTPUT.) Values for INPUT and OUTPUT 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. The only effect of INPUT and OUTPUT 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 (that is, 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 form. PPMFQQ then effectively puts those characters at the start of the file INPUT. The identifier of the parameter is passed to both routines (PPMFQQ and PPMUQQ). The identifier is used as a prompt.

The use of program parameters can best 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 PPMFQQ (CONST S : STRING); EXTERN;

PROCEDURE PPEFQQ;

PROCEDURE ALPHA [PUBLIC];
<declarations>
{Including those for P1, P2, .. Pn}
BEGIN
  PPMFQQ ('P1');  READFN (INPUT, P1);
  PPMFQQ ('P2');  READFN (INPUT, P2);
  .
  .
  PPMFQQ ('PN');  READFN (INPUT, Pn);
  PPEFQQ;
  {Called after all parameters are read}
  <program statements>
END;
```

See Section 19, "Run Time and Debugging," for more information on the routine PPMFQQ.

For program parameters of type FILE, the parameter in the command form is the file name. For other parameters, the value of the program variable is initialized to the value read from the command form. The program parameters INPUT and OUTPUT are exceptions; no parameter in the command form affects them.

If program parameters other than the special parameters INPUT and OUTPUT are used, note that you can supply the parameters in the Run command form or the program must be installed as an Executive command with the corresponding parameters. For example, consider a program that types a file to the video display using the file name as input. It would be installed as an Executive command with one field, "File name." The listing of such a program appears below.

Note that if an error occurs or if a program requires more parameters than appear on the command form, 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.

The example below illustrates the use of program parameters:

```
{   TYPE command - type a file to the video}
{display.  Accept a file as program parameter;}
{open this file for reading.  Read 1 byte from}
{the input file and echo to the video display.}
{Loop until end of file on input.}

  PROGRAM TypeFile (output, inFile);
  VAR
    b : BYTE;
     {file to type to the video display}
    inFile:  FILE of BYTE;

  BEGIN
      {open the input file}
    Reset(inFile);
      {ready to begin typing}
    WriteLn('Typing .. ');

      {loop until EOF on input}
     WHILE NOT EOF(inFile) DO
       BEGIN
          {read 1 byte from the input}
         Read(inFile, b);
          {echo this byte to the video display}
         Write(chr(b));
       END; {end while}

      {finished typing}
    WriteLn;
    WriteLn('DONE');

  END. {end TypeFile}
```

This example assumes the program was invoked with a command form such as:

```
Type
  File name_____________________________________
```

(For details of how to construct command forms, see the New Command command in the Executive Manual.)

Also note that the workstation operating system has more sophisticated parameter management facilities than those offered by the Pascal run-time, and can be used to access all subparameters entered in the command form, as well as the command name itself. (See the subsection "Parameter Organization" in Section 7, "Parameter Management," of the CTOS Operating System Manual for details.)

## MODULES

Modules provide a simple, straightforward method for combining several compilable segments into one program. Units, described in the next subsection, 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 the procedures and functions nested directly in the module. Depending on which, if any, attributes you specify, the following assumptions or restrictions apply:

- If there is no attribute list at all, the PUBLIC attribute is assumed. However, if a list is present but empty (for example, MODULE BETA [];) PUBLIC is not assumed.

- The EXTERN directive used with a particular procedure or function overrides the PUBLIC attribute given (or assumed) for the entire module.

- EXTERN and ORIGIN cannot be given as attributes for an entire module, although you can specify them for individual procedures and functions in a module.

- If PURE is used, the module must contain only functions for PURE.

Although a module contains no body, only declarations, you can use it as a parameterless procedure; that is, you can declare the module identifier as a procedure and call it from other programs, modules, or units. (If it is called from a procedure in the module itself, a declaration is not necessary.) A module procedure (unlike a similar procedure for programs or units) is never called automatically, since there is no way for the compiler to know whether a module has been loaded and thus whether to generate a call to it.

However, in some cases, the compiler generates module initialization code that should be executed by calling the module as an EXTERN procedure. If such code is necessary, the compiler gives the warning

```
Initialize Module
```

If you see this message, declare the module as a parameterless EXTERN procedure and call the procedure once before anything in the module is accessed. (You will need to do this if the 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;          {Run-time call}
  .           {initializes file}
  .           {variables.}
END.


MODULE M;
VAR F: FILE OF BYTE;
PROCEDURE USE_F;
.
.
END;
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 treated as program variables.

## UNITS

Units provide a structured way to access separately compiled modules. A unit has two parts:

Interface — Contains the declarations for the routines (procedures and functions), variables, and types, for example.

Implementation — Contains the actual code for the routine. Since the implementation does not define the routines it is implementing, it must textually include the interface.

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. The interface appears before the heading.

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 can 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.

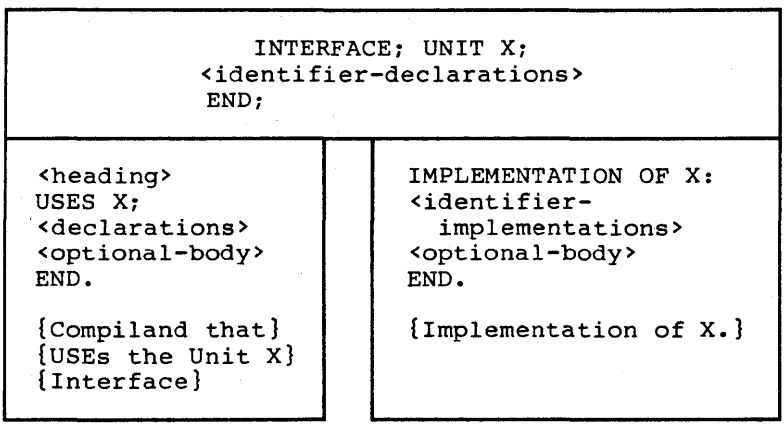



**Figure 16-1. A Unit.**

Note that if you use units, rather than modules, the run-time library is automatically linked to your program.

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 Figure 16-1, 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 can load a program with one of several implementations (for example, you could have one in Pascal or in assembly language). A large Pascal program is often better organized as a main program and a number of units. However, only a program, module, interface, or implementation can USE a unit, not an individual procedure or function.

A program, module, implementation, or interface that USES an interface must start with the source file for that interface. Generally, the interface source file is a separate file, and a $INCLUDE metacommand at the start of the source file brings in the interface source itself at compile time. Because there is then only one master copy of the interface, this is easier and more reliable than physically inserting the interface everywhere it is used (and running the risk of ending up with several different versions).

In some applications, you might want several versions of the same interface. 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 $INCLUDE the interface file at the start of the source file, as shown in Figure 16-2.

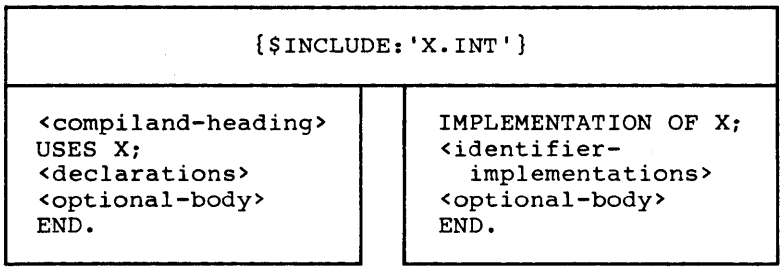



**Figure 16-2. Unit with File X.INT and a Compiland Using the Unit.**

If a procedure (or a variable) is declared in a module, then it has a PUBLIC attribute there and an EXTERN directive in any program that USEs it. This syntactical difference between the declarations makes it impossible to put the declarations in one file and $INCLUDE them into both compilands, as could be done with an interface. (Note that what is declared in an interface is not redeclared in the declaration section of the compilands that INCLUDE that interface.)

A source file of any kind contains zero or more unit interfaces, separated by semicolons, and followed by a program, a module, or an implementation, which is followed by a period. Each of these entities is called a "division." See the next subsections, "Interface Division," and "Implementation Division," for details about divisions.

A unit contains 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. These are exported by the interface. The unit is preceded by the keyword UNIT (for a provided unit) or USES (for a required one.)

All unit identifiers in a source file must be unique. The identifiers in parentheses, however, can differ in the providing and requiring divisions. That is, a compiland that uses a procedure from a unit can use a procedure name different from the one supplied by the interface. In this case, the name matching is done via the identifier lists. For example:

```
INTERFACE;
  UNIT V (P, Q, R);
  {List of identifiers declared in the unit.}
  .
  .
END;

PROGRAM TEST (INPUT, OUTPUT);
{Program using V}
USES V(A, B, C);
{The above list is matched against the
interface's list A = P, B = Q, C = R.}
BEGIN
  A; {P is called}
  B; {Q is called}
  C; {R is called}
END;
```

The identifier list in a USES clause is optional; if not given, the identifiers in the UNIT list are used by default. In that case, a compiland using a unit must use the same names for the unit's procedures and variables as are used in the interface. 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.

Example of a unit that introduces initialization code:

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 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;
```

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 implementation file:

```
{$INCLUDE:'GRAPHI'}
{$INCLUDE:'BASEPL'}
{The following implementation USES}
{the UNIT BASEPL.  Thus, the interface}
{is included above and the unit}
{used below.}
IMPLEMENTATION OF GRAPHICS;
{Implementation is invisible to user.}
  USES BASEPLOT;
    {Procedures BJUMP and WJUMP are}
    {implemented below.}
    {Note that only the identifiers}
    {are given in the heading.}
    {The parameter lists are given}
    {in the interface.}
  PROCEDURE BJUMP;
    BEGIN DRAWLINE (BLACK, X, Y) END;
  PROCEDURE WJUMP;
    BEGIN DRAWLINE (WHITE, X, Y) END;
BEGIN
{Begin initialization.}
  DRAWLINE (BLACK, Ø, Ø)
END.
```

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 for variables, procedures, and functions are associated with the corresponding identifiers in the interface, which then become external references for the Linker.

If the sample program and implementation above were compiled, every reference to the procedure PLOT would generate an external reference to WJUMP. However, references to DRAWLINE would use the same identifier for the external reference.

Constants and types (including any super array types) in the interface are simply entered in the program's symbol table (along with the new identifier, if any). Thus, a type in an interface is identical to the corresponding type in the USES clause.

Record field identifiers are the same in the program, interface, and implementation. Enumerated type constant identifiers must be given explicitly, if needed; they are not automatically implied by the enumerated type identifier. Labels cannot be provided by an interface, since the target label of a GOTO must occur in the same division as the GOTO.

**INTERFACE DIVISION**

The structure of an interface is as follows:

- An interface section starts with the reserved word INTERFACE, an optional version number in parentheses (assumed to be Ø if not given), and a semicolon.

- Next comes the keyword UNIT, the unit identifier, the list of exported (constituent) identifiers in parentheses, and another semicolon.

- Any other units required by this interface come next, in USES clauses.

- 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.

- The interface ends with BEGIN END and a semicolon, if it has initialization, or just with END and a semicolon, 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 depending whether the interface is given with the implementation of that unit (PUBLIC is used in that case) or in a compiland using the unit (EXTERN), you must not specify attributes that conflict (for example, 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.}
  {No code is permitted here.}
  END;
```

An exported identifier is used by programs and modules that use the same interface, rather than within the interface only.

In the example above, 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.

If you get this message you can reduce the number of identifiers in interfaces USEd by other interfaces. For example, make a single interface that contains only types (and type-related constants) shared by your other interfaces, and only USE this interface in the others.

If you include any file variables in the interface, the unit must be initialized. The compiler does not give the usual warning,

    Initialize Variable

when you declare a file in an interface. If your interface contains files, be sure to end it with BEGIN END so that it will be initialized.

**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:

- An implementation of an interface starts with the reserved words IMPLEMENTATION OF, followed by the unit identifier and a semicolon.
- Next comes a USES clause for units it needs for its own use, only.
- Then comes the usual LABEL, CONSTANT, TYPE, VAR, and VALUE sections and all procedures and functions mentioned as constituents (which must be declared at the higher level) 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;
    {No parameters or attributes here; they}
    {go into the interface.}
    BEGIN
      .
      {Code for FINDKEY}
      .
    END;

  PROCEDURE INKEY;
    BEGIN
      .
      {Code for INKEY}
      .
    END;

  PROCEDURE DELKEY;
    BEGIN
      .
      {Code for DELKEY}
      .
    END;

BEGIN
  .
  {Any user's initialization code goes here.}
  {Internal initialization code is}
  {produced automatically by the compiler.}
  {It is not placed here by the user}
  .
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 defined in the interface should generate public definitions for the Linker. You can also implement units in other programming languages, such as FORTRAN, or in a mixture of languages. If the interface is not implemented in Pascal, it must give the proper calling sequence attribute. (You must be familiar with calling sequences and internal representation of parameters).

Several Pascal run-time units are implemented partially in Pascal and partially in assembly language. As mentioned, any IMPLEMENTATION section that does not implement all interface procedures and functions must, at the start of the IMPLEMENTATION, declare such procedures and functions to be EXTERN.

An implementation, like a program, can 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 unit name appears in the USES clause 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 in another compiland.

# 17 METACOMMANDS

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

- debugging and error handling
- optimization level
- use of the source file during compilation
- listing file format

Metacommands are given within comments. 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, the following 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 most compiler directives during the course of a program. For example, the metacommand $LIST controls whether or not a listing is generated. Most of a program might use $LIST- (no listing generated), with a few sections using $LIST+ as needed. However, some metacommands, such as $LINESIZE, normally apply to an entire compilation.

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

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

- Typeless metavariables are invoked when used, as in $SIMPLE.

- Integer metavariables can be set to a numeric value, as in $PAGE:101.

- 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 if off, as in $MATHCK:1. Generally - is used to set the switch to off, as in $MATHCK-, and + is used to set it to on, as in $MATHCK+.

- 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. |
| :<n> | Metacommand is an integer. |
| + or - | Metacommand is an on/off switch.<br>+ sets value to 1 (on).<br>- sets value to 0 (off).<br>Default is indicated by + or - in heading. |
| :'<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 change to their ORD values. Thus, a Boolean FALSE value becomes Ø and TRUE becomes 1.

A complete alphabetic listing of Pascal metacommands is given in Table 17-2 and each command is discussed in detail further on in this section.

**Table 17-2. Metacommands. (Page 1 of 3)**

| Metacommand Name | Function |
|---|---|
| $BRAVE | Sends error messages and warnings to the video display. |
| $DEBUG | Turns on or off all the debug checking. |
| $ENTRY | Generates procedure entry/exit calls for the Pascal error handling routines. |
| $ERRORS | Sets the number of errors allowed per page. |
| $GOTO | Flags GOTO statements as "considered harmful." |
| $INCLUDE | Switches compilation from the current source file to the source file named. |
| $INCONST | Allows interactive setting of constant values at compile time. |
| $INDEXCK | Checks that array index values are in range, including super array indexes. |
| $INITCK | Checks for uninitialized values. |
| $IF $THEN $ELSE $END | Allows conditional compilation of source. |

**Table 17-2. Metacommands. (Page 2 of 3)**

| Metacommand Name | Function |
|---|---|
| $INTEGER | Sets the length of the INTEGER type. |
| $LINE | Generates line number calls for Pascal error processing routines. |
| $LINESIZE | Sets listing width. |
| $LIST | Turns on or off the source listing. |
| $MATHCK | Checks for mathematical errors such as overflow and division by zero. |
| $MESSAGE | Allows the display of a message on the video display at compilation time. |
| $NILCK | Checks for bad pointer values. |
| $OCODE | Turns on disassembled object code listing. |
| $PAGE | Sets the page number for the next page. |
| $PAGEIF | Skips to the next page if less than a specified number of lines are left on the page. |
| $PAGESIZE | Sets the length of a listing in lines. |
| $PUSH | Saves the current value of all metacommands. |
| $POP | Restores the saved value of all metacommands. |

**Table 17-2. Metacommands. (Page 3 of 3)**

| Metacommand Name | Function |
|---|---|
| $RANGECK | Checks for subrange validity. |
| $REAL | Sets the length of the REAL type. |
| $ROM | Gives a warning on static initialization. |
| $RUNTIME | Determines the context of run-time errors. |
| $SIMPLE | Disables global optimizations. |
| $SIZE | Minimizes the size of code generated. |
| $SKIP | Skips a specified number of lines or skips to end of page. |
| $SPEED | Minimizes the execution time of code. |
| $STACKCK | Checks for stack overflow at procedure or function entry. |
| $SUBTITLE | Sets the page subtitle. |
| $SYMTAB | Sends the symbol table to the listing file. |
| $TITLE | Sets the page title. |
| $WARN | Gives a warning message in the listing file. |

## OPTIMIZATION LEVEL

The metacommands shown in Table 17-3 let you control the degree to which optimization is used.

**Table 17-3. Optimization Level.**

| Name | Description |
|---|---|
| $INTEGER:<n> | Sets the length of the INTEGER type (default is 2.) |
| $REAL:<n> | Sets the length of the REAL type. |
| $ROM- | Gives an error on static initialization. |
| $SIMPLE | Disables global optimizations. |
| $SIZE | Minimizes the size of code generated. $SIZE is the default setting. |
| $SPEED | Minimizes the execution time of code. |

$INTEGER and $REAL set the length (that is, precision) of the standard INTEGER and REAL data types. $INTEGER can only be set 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 $SIMPLE metacommand turns off common subexpression optimization. $SIZE and $SPEED currently turn it back on again. $SIZE, $SPEED, and $SIMPLE are all mutually exclusive. The default is $SIZE.

If $ROM is set, the compiler gives an error message that static data are initialized in either of the following situations:

- at a VALUE section
- every place where static data initialization occurs due to $INITCK (described in the subsection "Debugging and Error Handling.")

## ERROR HANDLING AND DEBUGGING

The metacommands shown in Table 17-4 are for error handling and debugging using Pascal error handling routines. They also generate code to check for run-time errors.

Note that debugging in this context refers to Pascal's own error handling routines and not to the use of the programming tool, the Debugger, which is available as a part of the standard software for your workstation.

If any check is on when the compiler processes a statement or when the program executes it at run-time, 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:

- an error message
- a standard error code
- an optional error status value, such as an operating system return code

EMSEQQ also has available:

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

**Table 17-4. Error Handling and Debugging.**

| Metacommand | Description |
|---|---|
| $BRAVE+ | Sends error messages and warnings to the video display. |
| $DEBUG- | Turns on or off all the debug checking (those with CK suffix below and $ENTRY.) |
| $ENTRY- | Generates procedure entry/exit calls for Pascal error handling routines. |
| $ERRORS:<n> | Sets the number of errors allowed per page (default is 25). |
| $GOTO- | Flags GOTO statements as "considered harmful." |
| $INDEXCK- | Checks that array index values are in range, including super array indexes. |
| $INITCK- | Checks for the use of uninitialized values. |
| $LINE- | Generates line number calls for the Pascal error handling routines. |
| $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 the context of run-time errors. |
| $STACKCK- | Checks for stack overflow at procedure or function entry. |
| $WARN+ | Gives a warning message in the listing file. |

Each of these metacommands is discussed in more detail on the following pages:

| | |
|---|---|
| $BRAVE+ | Sends error messages and warnings to the video display (in addition to writing them to the listing file). |
| $DEBUG- | Turns on or off all the debug switches; INDEXCK, INITCK, MATHCK, NILCK, RANGECK, STACKCK. It is useful to use $DEBUG- at the beginning of a program to turn all checking off and then selectively turn on only the debug switches you want. Alternatively, you can use this metacommand to turn all debugging on at the start and then selectively turn off those you do not 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 the 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 can be useful in either of the following circumstances:

- o to flag all GOTO statements during the process of debugging
- o to encourage structured programming in an educational environment

$INDEXCK- Generates code to check that array index values, including super array indexes, are in range. Since array indexing occurs so often, bounds checking is enabled separately from other subrange checking.

$INITCK- Generates code to check for the occurrence of uninitialized values, such as the following:

- o uninitialized INTEGERs and 2-byte INTEGER subranges with the hexadecimal value 16#8000
- o uninitialized 1-byte INTEGER subranges with the hexadecimal value 16#80
- o uninitialized pointers with the value 1 (if $NILCK is also on)
- o uninitialized REALs with a special value

The $INITCK metacommand generates code to

- o set such values uninitialized when they are allocated
- o set the value of INTEGER range FOR-loop control variables uninitialized when the loop terminates normally
- o 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:

- variables mentioned in a VALUE section
- variant fields in a record
- components of a super array allocated with the NEW procedure

$LINE- Generates a call to the error handler for each source line of executable code. This allows the error handler to determine the number of the line in which an error has occurred. Because this metacommand generates a substantial amount of extra code for each line in a program, you should turn it on only when debugging. Note that $LINE+ requires $ENTRY+, so $LINE+ turns on $ENTRY, and $ENTRY- turns off $LINE.

$MATHCK- Generates code to check for mathematical errors, including INTEGER and WORD overflow and division by zero. $MATHCK does not check for an INTEGER result of exactly -MAXINT-1 (#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, "Available Procedures and Functions," for a description of each of these functions.)

$NILCK- Generates code to check for the following conditions:

- o dereferenced pointers whose values are NIL
- o uninitialized pointers if $INITCK is also on
- o pointers that are out of range
- o 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- Generates code to check subrange validity in the following circumstances:

- o assignment to subrange variables
- o CASE statements without an OTHERWISE clause
- o actual parameters for the CHR, SUCC, and PRED functions
- o indexes in PACK and UNPACK procedures
- o set and LSTRING assignments and value parameters
- o 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 video display, but you could link in a custom version of EMSEQQ, the error message routine, to do something different (such as invoke the run-time Pascal error handling routines or reset a controller). For more information on error handling, see Section 19, "Run Time and Debugging."

$STACKCK- Checks for stack overflow when entering a procedure or function and when pushing parameters larger than 4 bytes on the stack. Stack overflow is never checked in procedures with the INTERRUPT attribute.

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

## SOURCE FILE CONTROL

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 17-5 and described in more detail below.

**Table 17-5. Source File Control.**

| Name | Description |
|---|---|
| $IF <constant><br>$THEN <text1><br>$ELSE <text2> | Allows conditional compilation of <text1> source if <constant> is greater than zero. |
| $INCLUDE:'<filename>' | Switches compilation from the current source file to the source file named. |
| $INCONST:<identifier> | Allows interactive setting of constant values at compile time. |
| $MESSAGE:'<text>' | Displays message on the video to indicate which version of a program is compiling. |
| $PUSH | Saves the current value of all metacommands. |
| $POP | Restores the saved 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 = Ø;
{$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 can be a literal number or constant identifier. The text between $THEN, $ELSE, and $END is arbitrary; it can include line breaks, comments, other metacommands (including nested $IFs), etc. Any metacommands within skipped text are ignored, except, of course, corresponding $ELSE or $END metacommands.

Examples using the metaconditional:

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

$INCLUDE

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

$INCONST

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

```
Inconst:  YEAR =
```

Enter the desired parameter, for example, 1984:

```
Inconst: Year = 1984
```

The response is presumed to be of type WORD. The effect is to declare a constant identifier named YEAR with the value 1984. This interactive setting of the constant YEAR is equivalent to the constant declaration:

```
CONST YEAR = 1984;
```

except that the metacommand can be given anywhere in the program.

$MESSAGE

Allows you to send messages to the video display 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 that you can store with $PUSH and invoke with $POP. $PUSH and $POP are useful in $INCLUDE files for saving and restoring the metacommands in the main source file.

## LISTING FILE CONTROL

The metacommands listed in Table 17-6 and described in this subsection allow you to format the listing file as you wish. Listing file format, itself, is discussed in the subsection by that name.

**Table 17-6. Listing File Control Metacommands.**

| Metacommand | Description |
|---|---|
| $LINESIZE:<n> | Sets the width of listing. Default is 131. |
| $LIST+ | Turns on or off the source listing. Errors are always listed. |
| $OCODE+ | Turns on the disassembled object code listing. |
| $PAGE+ | Skips to the next page. Line number is not reset. |
| $PAGE:<n> | Sets the page number for the next page (does not skip to next page). |
| $PAGEIF:<n> | Skips to the next page if less than n lines are left on the current page. |
| $PAGESIZE:<n> | Sets the page length of a listing in lines. Default is 55. |
| $SKIP:<n> | Skips n lines or to the end of page. |
| $SUBTITLE:'<text>' | Sets the page subtitle. |
| $SYMTAB+ | Sends the symbol table to the listing file. |
| $TITLE:'<text>' | Sets the page title. |

$LINESIZE:<n> Sets the maximum length of lines in the listing file. Default is 131.

$LIST+ Turns on the source listing. Except for $LIST-, metacommands themselves appear in the listing. The format of the listing file is described in the subsection, "Listing File Format," below.

$OCODE+ Turns on the symbolic listing of the generated code to the object listing file. This listing looks like an assembly listing, with code addresses and operation mnemonics. The symbolic listing will not be generated unless you specify a file name in the [Object list file] field of the Pascal command form.

$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 the subsection, "Listing File Format"). The left columns contain the following:

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

A leading plus or minus sign indicates the sign of a frame offset.

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). It also contains 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 17-7.

$TITLE:'<title>'
Sets the name of a title that appears at the top of each page of the source listing.

**Table 17-7. 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 |

## LISTING FILE FORMAT

Listing file format is controlled by including various metacommands within the program.

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

```
Page   1

Ø6/25/84

11:21:24
 JG IC  Line#   Pascal 9.Ø
    ØØ      1

            2
    ØØ      3   PROGRAM foo;  {$symtab+}
    1Ø      4   VAR i:integer; k:ARRAY [-9..Ø] OF
                integer,
            4   -------------------------Warning
                156 , Assumed ;^
    2Ø      5   FUNCTION bar (VAR j: integer):
                integer;
    2Ø      6   VAR k: ARRAY [Ø..9] OF integer;
    2Ø      7   BEGIN
 +  21      8   6GOTO 1;{jump forward}
            8   ^Warning 281 Label Assumed
                Declared
            8   ^Warning 173 Insert :
            8   -------------^Warning 281 Label
                Assumed Declared
  = 21      9   i := bar (j);{assign to global}
    21     1Ø   1:{label}
  / 21     11   j := bar (i);{global to VAR parm}
  - 21     12   GOTO 1;{jump backward}
  * 21     13   RETURN; GOTO 1;{other jumps}
  % 21     14   i := bar (i);{other global
                reference}
    21     15   j:= bar (j);{no global references}
    1Ø     16   END;
           16   ----------^3Ø6 Function Assignment

                Not Found
```

```
Symtab     16   Offset Length   Variable - BAR
                -    2     24   Return offset,
                                  Frame length
                -    2      2   (function return)
                                  :Integer
                +    4      2   J : Integer VarP
                -   22     20   K :Array
           17
   10      18   BEGIN
   11      19    i := bar (i);
   00      20   END.

Symtab     20   Offset Length   Variable
                   0     50    Return offset,
                                  Frame length
                 28      2   I :Integer Static
                 30     20   K :Array   Static

                Errors  Warns  In Pass One
                    1      4
```

The listing file above is created when the sample program shown below is compiled:

```
    PROGRAM foo;  {$symtab+}
       VAR i:integer; k:ARRAY [-9..0] OF integer,
       FUNCTION bar (VAR j: integer): integer;
       VAR k: ARRAY [0..9] OF integer;
       BEGIN
    6  GOTO 1;{jump forward}
       i := bar (j);{assign to global}
       1:     {label}
       j := bar (i);{global to VAR parm}
       GOTO 1;{jump backward}
       RETURN; GOTO 1;{other jumps}
       i := bar (i);{other global reference}
       j:= bar (j);{no global references}
       END;

       BEGIN
        i := bar (i);
       END.
```

Every page of the listing file has a heading that includes such information as your title and subtitle, set with the metacommands $TITLE and $SUBTITLE, respectively. If these metacommands appear on the first source line, they take effect on the first page. The page number appears in the right side of the first line of the heading. The date and time appear in 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:

The JG column Contains flag characters generated for your information. Jump flags, which appear under the J, can 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

The IC column Contains information about the current nesting levels.

The digit under "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 shown in the C column refers to the control statement level; this number changes with BEGIN and END, CASE and END, and REPEAT and UNTIL pairs.

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.

The number in this column is useful for finding missing END keywords.

If a line does not contain any code processed 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.

The Line# column

Shows the line number of the line in the source file. A $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 warnings only can generate code, but the code may be bad. Warnings start with the word "Warning" and a number (see, for example, line 4 in the sample listing). Errors start with an error number (see line 16 in the sample listing). See Appendix A, "Compiler Error Messages," for a complete list 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 the video display (as well as to the listing file). However, if there are more than can fit on a single screen, the first ones scroll off.

The location of the error is indicated in the listing file with a caret (^). The message itself can appear to the left or right of the caret and is preceded by 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 can be displaced if the following line has tabs.

If the compiler encounters an error from which it cannot recover, it gives the message "Compiler Cannot Continue!". This message appears if any of the following occur:

- The keyword PROGRAM (or IMPLEMENTATION, INTERFACE, or MODULE) is not found, or the program, module, or unit identifier is missing.

- The compiler encounters an unexpected end-of-file.

- The compiler finds too many errors; the maximum number of errors per page is set with the $ERRORS metacommand (the default is 25).

- The identifier scope becomes too deeply nested. The maximum level to which procedures can be statically nested is 15.

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.

# 18 USING THE PASCAL COMPILER

You run a Pascal program by first compiling its one or more source modules, using the Linker to link the resulting object files with the Pascal library, and invoking the resulting run file. The run file is usually invoked through the Executive.

The Pascal compiler translates your Pascal source programs into object modules. The compiler provides a source listing, error messages, and a number of compiler metacommands to aid in program development and debugging.

The compiler comes with a set of object libraries to be linked with your code. These libraries provide complete run-time support for input/output, arithmetic functions, and inline code execution by the optional 8Ø87 Numeric Data Processor that is available with some workstations. When you link your program, the Linker automatically accesses these libraries when necessary. (The run-time libraries are discussed in Section 19, "Run Time and Debugging.")

Using the Linker, you can also combine Pascal object modules with those of other languages, for example FORTRAN, to facilitate writing applications that need different languages for different parts.

Pascal supports systems programming by providing access to all operating system services, such as direct (random) access to disk files, interrupt handling, and process creation. Calls also extend the range of services needed by the commercial application programmer: DAM, ISAM, Sort/Merge, and the Forms Run Time.

## COMPILING, LINKING, AND RUNNING PASCAL: OVERVIEW

To create and execute a Pascal program,

1. Create and edit the source file. You can use the Editor or the Word Processor to create the source file.

2. Compile the program. The compiler flags syntax errors as it reads your source file. You can place compiler controls called metacommands within your program to generate diagnostic calls for run-time errors. If compilation is successful, the compiler creates a relocatable object file.

3. Use the Linker to link compiled object files with the run-time library. A compiled object file is not executable and must be linked with one or more run-time libraries, using the Linker. Separately compiled subroutines in other languages or assembly language programs can also be linked to your program at this time. The Linker produces an executable file called a run file.

4. Use the Executive Run command to execute the resulting run file. (Alternatively, you can use the Executive command New Command to create a special command that you can use to execute your run file.)

Repeat this process until your program has successfully compiled, linked, and run without errors.

Since compiler metacommands can slow your program down, once the program runs without errors, remove or comment out any metacommands that are no longer necessary, then recompile, relink, and rerun your program.

## COMPILER OPTIONS

The compiler creates from your source file an object module, which must be linked to create a executable module, or run file.

In addition, the compiler creates a source list file (often referred to as the listing file), and you can optionally request that the compiler create an object list file.

The source list file gives the date and time of the compilation and a line-by-line account of the source file, with page headings and messages. Any error messages are also shown on your workstation screen. Appendix A "Compiler Error Messages" lists all the compiler error messages.

The various flags, level numbers, error message indicators, and symbol tables included in the listing file make it useful for error checking and debugging.

If you used the $INCLUDE metacommand to include other source files in the compilation, these files are also included in the source list file.

The listing file and its format is discussed more fully in Section 17, "Metacommands."

The object list file is a symbolic assembler-like listing of the object code. The addresses in the listing are relative to the start of the program or module.

The object listing file is used to

- check to see whether a different construct in assembly language would improve program efficiency
- provide a guide for debugging

Metacommands allow you to specify the form and content of your source code, object code, and output listing.

Controls are provided to copy source code from other files in addition to the main source file. The compiler also provides an optional symbol listing and controls to format the output listing to your own specifications. Using compiler metacommands is discussed in Section 17, "Metacommands."

## INVOKING THE COMPILER

You can compile Pascal source files by giving the Pascal command through the Executive on any workstation that has sufficient memory.

To invoke the compiler, type "Pascal" into the Executive command form. The Executive then displays the Pascal command form below:

```
Pascal
  Source file          _____________________________
  [Object file]        _____________________________
  [List file]          _____________________________
  [Object list file]   _____________________________
```

Source file — Enter the name of the source file you want to compile.

[Object file] — Enter the name you wish the compiler to give to the compiled object file.

If no name is specified, the compiler assigns a default name to the object file. The default name is created by removing from the source file name the last period (.) and any suffix following that period, then adding the suffix ".Obj". For example, for the source file [Dev]<Work>Program.1.Pas the compiler creates a default object file named [Dev]<Work>Program.1.Obj. If the source file does not have a suffix, then ".obj" is appended directly to the end of the file name.

[List file] — Enter the name of the list file to be created by the compiler. The list file is a listing of the source file and any warnings or error messages generated during compilation. If no list file name is specified, the default name used is the source file name with the suffix ".LST".

To list portions of the list file, see the $LIST metacommand.

[Object list file]

Enter the name you wish the compiler to use for the listing of the generated object code. If no file name is specified, the object list file is not generated.

To list only portions of the object list file, see the $OCODE metacommand.

After you have completed the Pascal command form, press GO and compilation begins.

The program is compiled in three passes. These are discussed in detail in the subsection "Compiler Structure" below.

After the compiler completes Pass One the following message is displayed.

```
Pass One   No Errors Detected
```

If the compiler detects errors during compilation, messages such as the following appear:

```
Pass One   2 Warnings Detected
Pass One   3 Errors Detected
```

The error and warning messages also appear on your list file.

An error is a mistake that would prevent the program from running correctly and stops the compilation.

A warning indicates a condition that will not prevent the program from running, but which can produce invalid results or can be poor programming practice.

See Appendix A, "Compiler Error Messages," for a complete listing of messages and information about how to correct the errors in your program.

Pass Two of the compiler produces the object file. When it is complete a message similar to the one below is displayed:

```
Code Area Size = #Ø5EC     ( 1516)
Cons Area Size = #ØØE6     (  23Ø)
Data Area Size = #Ø264     (  612)

Pass Two   No Errors Detected
```

The first three lines indicate, first in hexadecimal and then in decimal notation, the amount of space taken by executable code (Code), constants (Cons), and variables (Data). The number of errors given is for Pass Two only.

The third pass produces the object list file and is executed only if you request one.

For a more detailed discussion of the compiler see the subsection "Compiler Structure" below.

## LINKING A PASCAL PROGRAM

The Linker is invoked through the Executive, by typing "Link" (or as many letters as required to make the command unique) into the Executive command form. The following form is displayed:

```
Link
  Object modules          _____________________
  Run file                _____________________
  [List file]             _____________________
  [Publics?]              _____________________
  [Line numbers?]         _____________________
  [Stack size]            _____________________
  [Max memory array size] _____________________
  [Min memory array size] _____________________
  [System build?]         _____________________
  [Version]               _____________________
  [Libraries]             _____________________
  [DS Allocation]         _____________________
  [Symbol file]           _____________________
```

Using the Linker and completing each of the fields of the Link form are discussed in detail in the Linker/Librarian Manual. The following special features of the Linker are important for use with Pascal:

Object modules

Enter the name(s) of the object modules you want linked. Leave a space between each object file specification. (If you have too many entries to fit on the command line, you must place the entries in a file, called an at-file, then place the file name on the command line prefixed by an at sign (@). The use of at-files is discussed in the Executive Manual.)

If your program includes run-time overlays, you must include the file [Sys]<Sys>PasSwp.obj in this field.

If your program includes floating point calculations and you have an 8087 chip installed on your workstation, you can include the entry @[Sys]<Sys>Pascal8087.fls. (If you do not have an 8087 chip installed

on your workstation and your program uses floating point constants or variables, the 8087 emulator is automatically linked to your programs.)

If you are linking a minimal Pascal program, include the file [Sys]<Sys>Pasmin.obj. Minimal Pascal is discussed in the subsection "Avoiding the Run-time Library" in Section 19, "Run time and Debugging."

[Stack size] Default for Pascal: 8K

If you wish to change stack size, specify the desired size here.

[Libraries] When linking a Pascal program, the Linker automatically searches the library [Sys]<Sys>Pascal.Lib (if it exists) for any unresolved external symbols. The library [Sys]<Sys>CTOS.Lib is also searched.

You can specify any additional libraries you wish, for example, if you are linking with subprograms written in other languages, then the libraries for those languages must be specified.

If you are linking a program from which you wish to exclude any references to the Pascal run-time library, it is recommended that you specify the libraries that you do wish to link with your program followed by the word none. For example,

[Libraries] [Sys]<Sys>CTOS.Lib none

In this case, if any calls are made to the run time, the Linker indicates an unresolved external.

[DS Allocation?] Default for Pascal: Yes

This field is used to minimize the run-time value of DS (the data segment register) by offsetting all references to group DGroup.

Group DGroup consists of 64K bytes or less allocated for constants, data, and stack.

If you specify "Yes", the default, then the entire 64K bytes can be used for your data if necessary. The Code Segment is loaded at the high end of memory, above the data segment. Relative addressing starts at the highest word, no matter how much space is really needed for DGroup.

If you specify "No," however, the data segment takes only the amount of space actually needed and is loaded at the high end of memory, with the Code Segment below it.

For example, if your program uses only 32K of data and you specify "Yes" for [DS Allocation?] then the address of DS is DS:FFFF, whereas if you specify "No" the address is DS:$0032K. Figure 18-1 illustrates this.

Most Pascal applications require [DS Allocation?] to be "Yes." Object module procedures and tasks produced by the Pascal compiler use a single value in DS during their entire execution, and include the group DGroup with DS equal to DGroup. This feature must be used for linking Pascal tasks that make use of the Pascal heap.

Run files linked using Pasmin.obj can have DS Allocation set to either "Yes" or "No."

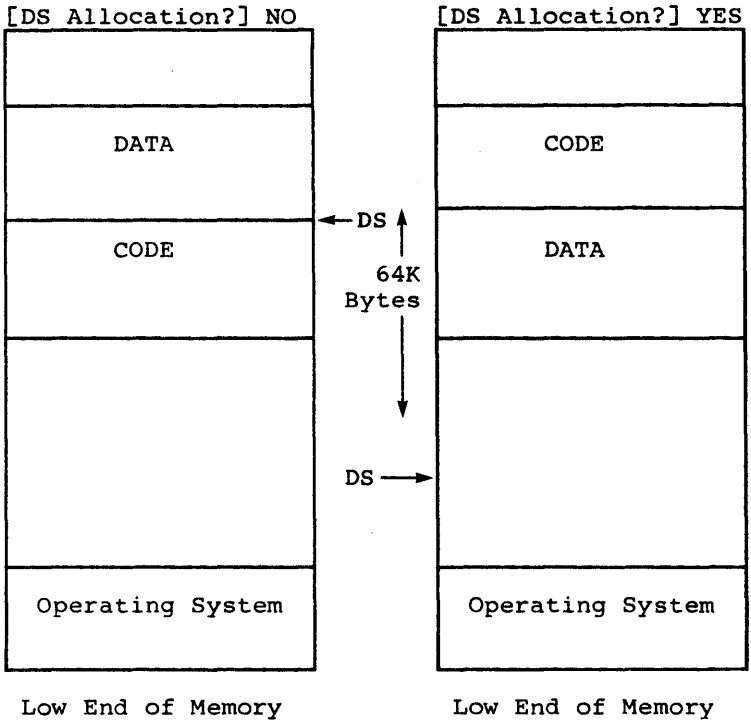



**Figure 18-1. DS Allocation.**

## RUNNING A PASCAL PROGRAM

Once a run file has been obtained through use of the Linker, a Pascal program can be run either by using the Executive's Run command, or by creating a customized command, using the New Command command. Parameters can be passed to the Pascal program parameters declared in the program header whether you create a customized command form or use the Run command.

(See Section 16, "Compilable Parts of a Program" for information on passing program parameters from the command form to the program.)

### EXAMPLE

The sample program below writes the contents of two fields to the video display. The run file for the program has the name Write.Run.

```
Program ReadParam(OUTPUT, field1, field2);

VAR   field1, field2 : LSTRING (255);

BEGIN

      WriteLn(field1);
      WriteLn(field2);

END.
```

If you run the program by completing the Executive Run command form as shown below, and the words "this" and "that" are written to the video display, "this" and "that" are the parameters for field1 and field2, respectively.

```
Run
  Run file          Write.Run
  [Case]
  [Parameter 1]     this
  [Parameter 2]     that
  [Parameter 3]
  [Parameter 16]
```

To create a customized command to invoke Write.Run, you can complete the Executive New Command form as shown below.

```
New Command
  Command name        Write
  Run file            Write.Run
  Field names         'Field 1' 'Field 2'
  Description         'Write 2 fields to screen'
  [Overwrite ok?]
  [Case (default
      'ØØ')]
  [Command file]
```

Then when you type "Write" into the Executive command line and press RETURN, the following command form is displayed:

```
Write
  Field 1
  Field 2
```

After you complete the form and press GO, Write.run is invoked and the parameters you typed into the parameter fields Field 1 and Field 2 are written to the video display.

(Using the New Command command and the Run command is described in detail in Section 13, "Commands," of the Executive Manual and in the subsection "Adding a New Command" in Section 5, "Advanced Concepts," of the same manual.)

## COMPILER STRUCTURE AND MEMORY REQUIREMENTS

The structure of the compiler is described in this subsection for your information only. It is not necessary to understand this information to use the compiler.

The compiler is written in Pascal.

The compiler is divided into three phases, or passes, each of which performs a specific part of the compilation process.

Pass One, executed by [Sys]<Sys>PascalFE.Run performs the following actions:

- reads the source program
- translates the source into an intermediate code
- produces the source listing file
- produces the symbol table file
- produces the intermediate code file

Pass One creates two intermediate files, Pasibf.Sym and Pasibf.Bin. These incorporate information from your source file and from the file Paskey, which contains Pascal predeclarations. These two files are always written to your default directory.

Pass Two is performed by [Sys]<Sys>PascalOpt.Run and does the following:

- optimizes the intermediate code
- generates target code from intermediate code
- produces and reads an intermediate binary file
- produces the object (link text) file

Pass Two reads and then deletes Pasibf.Sym and Pasibf.Bin. Pass Two creates and, if you did not request an object listing file, later deletes the intermediate file Pasibj.Tmp. If you requested an object listing, the second pass also creates the intermediate file Pasibf.Oid.

The third pass, performed by the file [Sys]<Sys>PascalLst.Run, produces the object listing file and is only invoked if you specifically request an object listing when you complete the Pascal command form. During the third pass the files Pasibf.Tmp and Pasibf.Oid are deleted.

All intermediate files contain Pascal records. A common constant and type definition file is used called Pascom.nnm, which defines the intermediate code and symbol table types. A similar file is used during the second and third passes for the intermediate binary file definition.

The intermediate code (or ICode) record contains an ICode number, opcode, and up to four arguments; an argument can be the ICode number of another ICode to represent expressions in tree form, or another value, such as a symbol table reference, constant, or length. The intermediate binary code record contains several variants for absolute code or data bytes, public or external references, label references and definitions, etc.

The symbol table record is complex, with a variant for every kind of identifier (such as, assorted data types, variables, procedures, and functions.)

The compiler itself takes memory, and in addition needs memory for its internal tables. It puts some of these tables into the long heap, the others into the short heap. The long heap is limited only by the computer memory. Exact size of the compiler and memory requirements for the short heap and the compiler stack are detailed in the Pascal Release Notice.

A compilation can sometimes terminate abnormally on the first pass with the error message 'Compiler Out of Memory.' This message usually indicates stack/heap space overflow. Examples of information that is stored in the short heap are: PUBLIC and EXTERN declarations, and TYPE declarations. Reducing the number of the these declarations in your program, can help it to compile successfully. Note, however, that these declarations do not affect the size of the program when it runs. They only affect memory requirements during compilation.

## VIRTUAL CODE MANAGEMENT FACILITY

Pascal is compatible with the Virtual Code Management facility. The Virtual Code Management facility is described in detail in Section 6, "Virtual Code Segment Management," in the CTOS Operating System Manual.

As with all applications that use the Virtual Code Management facility, the swap buffer must be allocated and initialized before any overlay is called. You can overlay both portions of the Pascal run-time system and portions of your own program.

To include portions of the run-time system in overlays, the following are necessary:

- Include PasSwp.obj in the Object modules field of the Linker command form.
- Write a procedure called BEGOQQ to perform user initialization. These procedures must be included (in the Object Modules field of the Linker command form) when the Pascal application is linked.

Pascal provides an empty procedure, BEGOQQ as an entry point. You can use it to initialize a swap buffer before any Pascal run-time initialization takes place. You must allocate and initialize the swap buffer in BEGOQQ to ensure that the swap buffer is ready when the Pascal run-time system is invoked.

For example:

```
module misoqq[];

Type adsw = ads of word;

(* InitOverlays is a CTOS function
initializing a swap
 buffer.
*)

Function InitOverlays(pBuf:adsw;cb:word):word;
            extern;
Procedure CheckErc(erc:word); extern;
```

```
Var buf: array[1..10240] of word;
    (* Memory for a swap buffer of 10240
bytes.*)

(* The following procedure initializes the
above swap buffer.*)

procedure begoqq [public];
begin

CheckErc(InitOverlays(ads buf[1],10240))
end;
end.
```

The following modules, however, must always be resident:

```
Cmpd7Alt Comr7Alt Oemr7Alt Erre ErreeAlt
Emtr7Alt Emur7Alt Emus7Atl Heah Lscw7Alt
MishcAlt Misg6Alt Misy Pasmax Riauqq Ribuqq
Rndc7 TsdrAlt.
```

If the Pascal8087.Lib is used, its modules must also be resident.

If the Linker issues warnings regarding CALL/RET conventions for the modules listed above, such warnings can be ignored, because a procedure that is resident in memory and does not pass control to any nonresident procedure does not have to satisfy the CALL/RET conventions, and the set of procedures above satisfies this requirement.

(See the subsection "Virtual Code Segment Management and Assembly Code" in Section 9, "Accessing Standard Services from Assembly Code," in the Assembly Language Manual for a discussion of overlay conventions.)

# 19 RUN TIME AND DEBUGGING

The run-time support libraries contain object modules that can be linked to your program to satisfy unresolved external references. When your Pascal program is linked, the library files [Sys]<Sys>Pascal.Lib and [Sys]<Sys>CTOS.Lib are automatically searched and the appropriate modules are linked to it if necessary.

The run-time support libraries provide all input/output (I/O) support needed to run your programs on your system. If you choose to use floating-point software routines all required arithmetic and interface software is also provided by the run-time libraries. If you have the 8Ø87 chip installed on your system, you can specify a special library at link time to take advantage of this chip for your floating-point routines. (See Section 18, "Using the Pascal Compiler," for more information on how to link your program.)

## OVERVIEW OF THE PASCAL RUN TIME

Run-time routines linked to a Pascal program are described briefly below. Pascal run-time routines all have six character names and end in the suffix QQ. Run-time routines are discussed in detail in the subsection "Run-Time Architecture" below.

The run file produced by the Linker for a Pascal program has the entry point BEGXQQ, which is a routine written in assembly language. This routine sets the initial stack pointer, the starting address of the heap, and various other routine variables. There is also a call to initialize the Pascal file system. Finally, there is a call to the Pascal program, which is always given the name ENTGQQ.

The Pascal main program continues the initialization process. Every unit mentioned in a USES clause in any interfaces or in the program is initialized by calling it as a procedure, in the order of the USES clauses. Any files declared in the program are initialized by calling NEWFQQ for each one. Finally, any program parameters are read and assigned to their variables, and the actual program code begins.

When the program terminates, the call to ENTGQQ returns to procedure BEGXQQ, which calls ENDXQQ. The Pascal file system is then called to close all open files and to discard all temporary files. A call to Exit in the operating system terminates the program.

Inside a Pascal application, many calls are also made to the Pascal run time to accomplish tasks too complicated to be done by straight generated code. For example, most error checking is accomplished by calling run-time helpers. You can identify these calls by their names: all run-time routines have six character names ending in QQ.

Note that run-time routines are not reentrant. Therefore, if one application creates several processes that execute concurrently a piece of code written in Pascal, care must be taken that only one of them is executing Pascal run-time code at any one time.

All CTOS facilities are available for use from Pascal. Interfaces to routines are described in the CTOS Operating System Manual and examples of the use of the operating system from Pascal are given in this manual in Appendix F, "Using Pascal as a Systems Programming Language."

## DEBUGGING

Pascal programs may be run under the control of the Debugger. (Note that the term Debugger here does not refer to Pascal error handling routines, but to the Debugger available with the standard software for your workstation.) To pass control to the Debugger, use CODE-GO rather than GO when you invoke your program. Using the Debugger is described in detail in the Debugger Manual.

The use of symbol files and object list files is very helpful in the debugging of Pascal programs.

The symbol file gives you the addresses of public variables for your program. The symbol file is created by the Linker when your program is linked. The name of the symbol file has the extension ".Sym".

The entry point into the main program is ENTGQQ (a public variable).

The object list file is a symbolic assembler-like listing of the object code that lists addresses of the instructions relative to the start of the program or module.

The example below shows code from an object list file for the Pascal statement i := i+1; where i is an integer.

```
L5:
     **  000011      MOV         AX,I
     **  000014      INC         AX
     **  000015      MOV         I,AX
```

The L5 indicates that this statement is on line 5 of the program. The numbers on the left side of the code indicate the hexadecimal offset from the beginning of the code segment for the particular instruction. For example, the MOV AX,I instruction begins at CS:11, where CS is the current code segment address.

## RUN-TIME ROUTINES

The Pascal run-time entry point and variable names all have six characters, the last three of which consist of a unit identification letter followed by the letters "QQ".

Table 19-1 shows the current unit identifier suffixes.

**Table 19-1. Unit Identifier Suffixes. (Page 1 of 2).**

| Suffix | Unit Function |
|---|---|
| AQQ | Reserved |
| BQQ | Compile time utilities |
| CQQ | Encode, decode |
| DQQ | Double precision real |
| EQQ | Error handling |
| FQQ | Pascal file system (Unit F) |
| GQQ | Generated code helpers |
| HQQ | Heap allocator |
| IQQ | Reserved |
| JQQ | Reserved |
| KQQ | FCB definition |
| LQQ | STRING, LSTRING |
| MQQ | Reserved |
| NQQ | Reserved |

**Table 19-1. Unit Identifier Suffixes. (Page 2 of 2).**

| Suffix | Unit Function |
|---|---|
| PQQ | Reserved |
| RQQ | Real (single precision) |
| SQQ | Set operations |
| TQQ | Reserved |
| UQQ | Operating system file system |
| VQQ | Reserved |
| WQQ | Reserved |
| XQQ | Initialize/terminate |
| YQQ | Special utilities |
| ZQQ | Reserved |

## MEMORY ORGANIZATION

Memory on the CPU is divided into segments, each containing up to 64K bytes. The Linker also puts segments into classes and groups. All segments with the same class name are loaded next to each other. All segments with the same group name must reside in one area up to 64K bytes long; that is, all segments in a group can be accessed with one segment register.

Pascal uses the medium model of computation, that is, it uses multiple code segments, but only one data segment, called DGroup. Memory is allocated within DGroup for all static variables, constants that reside in memory, the stack, and the short heap.

DGroup is addressed using the DS (current data) or SS (current stack) segment register. Normally, DS and SS contain the same value, although DS may be changed temporarily to some other segment and changed back again. SS is never changed; its segment registers always contain abstract "segment values" and the contents are never examined or operated on. Long addresses, such as ADS variables, use the ES segment register for addressing.

Memory in DGroup is normally allocated from the top down; that is, the highest addressed byte has DGroup offset 65535, and the lowest allocated byte has some positive offset. This allocation means offset zero in DGroup may address a byte in the code portion of memory, in the operating system below the code, or even below absolute memory address zero. (In the latter case the values in DS and SS are "negative.")

DGroup has two parts:

- a fixed-length upper portion containing static variables and constants

- a variable-length lower portion containing the heap and the stack

After your program is loaded, during initialization (in ENTXQQ), the fixed upper portion is placed as high as possible to make room for the lower portion. If there is enough memory, DGroup is expanded to the full 64K bytes; if there is not enough room for this, it is expanded as much as possible.

Figure 19-1 illustrates memory organization as described above.

Note that memory organization appears differently than as shown in Figure 19-1, if, when you link your program, you set the field "[DS Allocation?] to "No." In that case the Data segment is not expandable and is loaded above the Code segment. (See the subsection "Linking Your Program" in Section 18, "Using the Compiler," for an explanation of DS Allocation.)

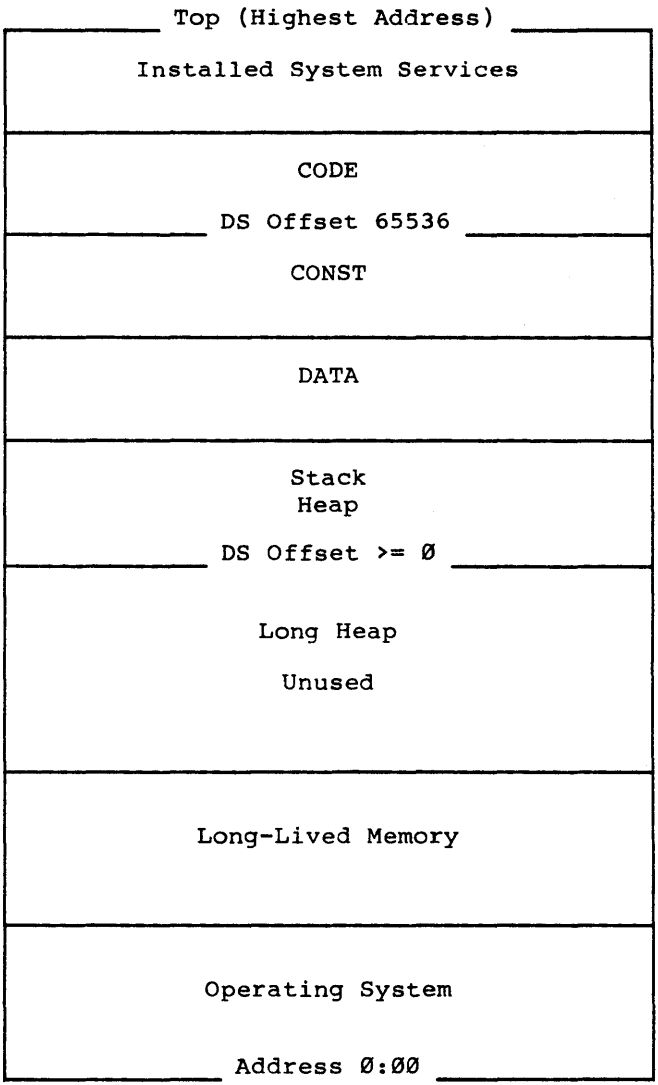



**Figure 19-1. Memory Organization, Single Partition Operating System.**

## INITIALIZATION AND TERMINATION

Every executable file contains one, and only one, starting address. As a rule, when object modules are involved, this starting address is at the entry point BEGXQQ in the module PASMAX. A program (as opposed to a module or implementation) has a starting address at the entry point ENTGQQ. BEGXQQ calls ENTGQQ.

The following discussion assumes that a main program along with other object modules is loaded and executed. However, you can also link a main program in assembly or some other language with other object modules. In this case, some of the initialization and termination done by the PASMAX module may need to be done elsewhere.

When a program is linked with the run-time library and execution begins, several levels of initialization are required. The levels, in the order in which they occur, are the following:

- machine-oriented initialization
- run-time initialization
- program and unit initialization

The general scheme is shown in Table 19-2.

**Table 19-2. Pascal Program Structure.**

**PASMAX module**

```
ENDXQQ: {Aborts come here}
        Call ENDOQQ
        Call ENDYQQ
        Call ENDUQQ
        Call ENDX87
        Exit to operating system

BEGXQQ: Set stack pointer, frame pointer
        Initialize PUBLIC variables
        Set machine-dependent flags,
        registers, and other values
        Call INIX87
        Call INIUQQ
        Call BEGOQQ
        Call ENTGQQ {Execute program}
        Call ENDXQQ {Termination}
```

**INTR module**

```
INIX87: Real processor initialization

ENDX87: Real processor termination
```

**UNIT U module**

```
INIUQQ: Operating system specific file unit
        initialization

ENDUQQ: Operating system specific file unit
        termination
```

**MISO module**

```
BEGOQQ: (Available for other user
        initialization procedures)

ENDOQQ: (Available for other user
         termination procedures)
```

**Program module**

```
ENTGQQ: Call INIFQQ
        If $ENTRY on, CALL ENTEQQ
        Initialize static data
        Initialize units
        FOR program parameters DO
          Call PPMFQQ
        Execute program
        If $ENTRY on, CALL EXTEQQ
```

**Machine Level Initialization**

The entry point of a load module is the routine BEGXQQ, in the module PASMAX. BEGXQQ does the following:

- Initializes constant and static variables. The initial stack pointer is put into PUBLIC variable STKBQQ and is used to restore the stack pointer after an interprocedure GOTO to the main program.

- Sets the frame pointer (that is, the pointer to the current procedure) to zero.

- Initializes a number of PUBLIC variables to zero or NIL. These include

  - RESEQQ, a machine error context

  - CSXEQQ, a source error context list header

    PNUXQQ, an initialized unit list header

  - HDRFQQ, an open file list header

- Sets machine dependent registers, flags, and other values.

- Sets the short heap control variables. BEGHQQ and CURHQQ are set to the lowest address for the heap: the word at this address is set to a heap block header for a free block the length of the initial heap. ENDHQQ is set to the address of the first word after the heap. (The initial heap is empty.) The stack and the heap grow together, and the PUBLIC variable STKHQQ is set to the lowest legal stack address (ENDHQQ, plus a safety gap).

  The long heap is initialized when the user calls a long heap routine.

- If the program uses REAL numbers, calls INIX87, the real processor initializer. This routine initializes an 8087 or sets 8087 emulator interrupt vectors, as appropriate.

- Calls INIUQQ, the file unit initializer. If the file unit is not used and you do not want it loaded, a dummy INIUQQ routine that only returns must be loaded. Pasmin.Obj provides an empty INIFQQ instead of calling INIUQQ.

- Calls BEGOQQ, the escape initializer. In a normal load module, an empty BEGOQQ that only returns is included. However, this call provides an escape mechanism for any other initialization. For example, it could initialize tables for an interrupt driven profiler or to initialize overlay management.

  If you want a nonempty initialization, you must writer your own BEGOQQ routine. (See Appendix F, "Using Pascal as a Systems Programming Language," for an example of a module that uses BEGOQQ to allocate and initialize a swap buffer.)

- Calls ENTGQQ, the entry point of your program.

- Calls ENDXQQ, the termination procedure.

**Program Level Initialization**

Your main program continues the initialization process. First, the file system, a parameterless procedure called INIFQQ, is called. If you link your program with Pasmin.Obj, an empty INIFQQ is provided.

After the file initialization, if the metacommand $ENTRY is on during compilation, ENTEQQ is called to set the source error context. Next, each file at the program level gets an initialization call to NEWFQQ.

After static data initialization comes unit initialization. Every USES clause in the source, including those in INTERFACEs, generates a call to the initialization code for the unit.

Units may or may not contain initialization code. If the interface contains a trailing pair of BEGIN and END statements, initialization code in the implementation is presumed. Units are initialized in the order that the USES clauses are encountered.

Finally, any program parameters are read or otherwise initialized, and your program begins. Except for INPUT and OUTPUT, PPMFQQ is called for each parameter to set the parameter's string value as the next line in the file INPUT. Then one of the READFN routines "reads" and decodes the value,

assigning it to the parameter. The parameter's identifier is passed to PPMFQQ for use as a prompt. PPMFQQ first calls PPMUQQ to get the text of any parameters from the command form. If PPMUQQ returns an error, then PPMFQQ does the prompting and reads the response directly.

User unit initialization is much like user program initialization. The following actions occur:

- error context initialization, if $ENTRY meta-command was on during compilation
- variable (file) initialization
- unit initialization for USES clause
- user unit initialization

Calls to initialize a unit can come from more than one unit. The unit interface has a version number, and each initialization call must check that the version number in effect when the unit was used in another compilation is the same as the version number in effect when the unit implementation itself was compiled. Except for this, unit initialization calls after the first one should have no effect; that is, a unit's initialization code should be executed only once. Both version-number checking and single, initial-code execution are handled with code automatically generated at the start of the body of the unit. This has the effect of

```
IF INUXQQ (useversion, ownversion, intrec,
           unitid)
THEN RETURN
```

The interface version number used by the compiland using the interface is always passed as a value parameter to the implementation initialization code. This is passed as "useversion" to INUZQQ. The interface version number in the implementation itself is passed as "ownversion" to INUXQQ. INUXQQ generates an error if the two are unequal.

INUZQQ also maintains a list of initialized units. INUXQQ returns true if the unit is found in the list, or else puts the unit in the list and returns false. The list header is PNUXQQ. A list entry passed to INUXQQ as "initrec" is initialized to the address of the unit's identifier (unitid) plus a pointer to the next entry.

User modules (and uninitialized implementations of units) may have initialization code, much like a program and unit implementation's initialization code, but without user initialization code or INUXQQ calls.

The initialization call for a module or uninitialized unit cannot be issued automatically. When the module is compiled, a warning is given if an initialization call is required (that is, if there are any files declared or USES clauses.) To initialize a module, declare the module name as an external procedure and call it at the beginning of the program.

**Program Termination**

Program termination occurs in one of three ways:

- The program may terminate normally, in which case the procedure ENDXQQ is called.
- The program may abort because of an error condition, either with a user call to ABORT or a run-time call to an error handling routine. In either case, an error message, error code, and error status are passed to EMSEQQ, which does whatever error handling it can and calls ENDXQQ.
- ENDXQQ can be declared as an external procedure and called directly.

ENDXQQ first calls ENDOQQ, the escape terminator, which normally just returns to ENDXQQ. Then ENDXQQ calls ENDYQQ, the generic file system terminator. ENDYQQ closes all open Pascal files, using the file list headers HDRFQQ and HDRVQQ. ENDXQQ calls ENDUQQ, the file unit terminator. Finally, ENDXQQ calls ENDX87 to terminate the real number processor (8087 or emulator.) As with INIUQQ and INIFQQ, if your program requires no file handling, you can declare empty parameterless procedures for ENDYQQ and ENDUQQ. The main initialization and termination routines are in module PASMAX. Procedure BEGOQQ is in the module MISOALT1; ENDUQQ is in RICUQQ; and ENDYQQ is in MISY.

**Using the Initialization and Termination Points in Your Program**

The routines BEGOQQ and ENDOQQ are provided by the run-time library as entry points for you to use. The program example that follows uses these entry points to display the date and time.

```
{$debug-}

Program UserInitAndTermination (Output);
    {This program sample describes how to use the
    initialization and termination entry points
    that the run time provides for the user.

    The nubs provided for initialization and
    termination are labeled 'BEGOQQ' and 'ENDOQQ'
    respectively.

    Since these entry points are defined by the
    run-time library, this compiland must be
    linked with Pascal.Lib.}


Type
    pbType = ads of word;
    DateTimeType = array [1..2] of word;
    ExpDateTimeType = array [1..4] of word;


Var [public]
    lsDateTime :lstring(30);
    DateTime   :DateTimeType;
    ExpDateTime:ExpDateTimeType;


{Definition of CTOS externals to be used:}

Var [extern]
    bsVid:array [1..130] of byte;
    {'bsVid' is an open video bytestream declared
    in CTOS.Lib}

Function FormatTime
    (plsDateTimeRet:pbType;
    pExpDateTime:pbType ) :word; extern;

Function GetDateTime
    (pDateTimeRet:pbType ) :word; extern;

Function ExpandDateTime
    (dateTime  :DateTimeType;
    pExpDateTime:pbType )   :word; extern;
```

```
Function WriteBsRecord (
    pBswa       :pbType;
    pbRec       :pbType;
    cbRec       :word;
    pbCbRet     :pbType)  :word; extern;

Procedure CheckErc
    (erc        :word );       extern;

Procedure BEGOQQ[public];
        var cbRet :word;
        begin
        {This procedure will be called by the run-
        time initialization.  It will display a
        banner with the date/time}
        CheckErc (GetDateTime (ads DateTime));
        CheckErc (ExpandDateTime (DateTime, ads
               ExpDateTime));
        CheckErc (FormatTime (ads lsDateTime, ads
               ExpDateTime));
        CheckErc (WriteBsRecord (ads bsVid,
               ads 'Program initialization at ',
               26, ads cbRet));
        CheckErc (WriteBsRecord (ads bsVid,
               ads lsDateTime[1], lsDateTime.len,
               ads cbRet));
        CheckErc (WriteBsRecord (ads bsVid, ads
               #Øa, 1, ads cbRet));
        end;

Procedure ENDOQQ[public];
    var cbRet :word;
    begin
        {This procedure will be called by the run-
        time termination before the first
        executable statement of the program. It
        will display a banner with the date/time.
        Note that if the CTOS calls 'Exit' or
        'ErrorExit' are used the run-time
        termination is circumvented.}
    CheckErc (GetDateTime (ads DateTime));
    CheckErc (ExpandDateTime (DateTime, ads
        ExpDateTime));
    CheckErc (FormatTime (ads lsDateTime, ads
        ExpDateTime));
    CheckErc (WriteBsRecord (ads bsVid, ads
        'Program termination at ', 23, ads
        cbRet));
    CheckErc (WriteBsRecord (ads bsVid, ads
        lsDateTime[1], lsDateTime.len, ads
        cbRet));
    CheckErc (WriteBsRecord (ads bsVid, ads #Øa,
        1, ads cbRet));
    end;
```

```
begin{start of program, after run-time
    initialization}
    Writeln;
    Writeln ('Hello');
    Writeln;
end.
```

**ERROR HANDLING**

Run-time errors are detected in one of four ways:

- The user program calls EMSEQQ (that is, ABORT).
- A run-time routine calls EMSEQQ.
- An error checking routine in the error module calls EMSEQQ.
- An internal helper routine calls an error message routine in the error unit which, in turn, calls EMSEQQ.

Handling an error detected at run-time usually involves identifying the type and location of the error and then terminating the program. The error type has three components

- a message
- an error number (Pascal error code)
- an error status code (operating system return code)

The message describes the error and the number can be used to look up more information. The error status value is undefined, although for file system errors it may be an operating system return code. However, the error status value may also be used for other special purposes. Table 19-3 shows the general scheme for error code numbering.

An error location has two parts:

- machine error context
- source program context

The machine error context is the program counter, stack pointer, and stack frame pointer at the point of the error. The program counter is always the address following a call to a run-time routine (for example, a return address.)

**Table 19-3. Error Number Classification.**

| Range | Classification |
|---|---|
| 1- 999 | Reserved for user ABORT calls |
| 1000-1099 | Unit U file system errors |
| 1100-1199 | Unit F file system errors |
| 1200-1299 | Unit V file system errors |
| 1300-1999 | Reserved |
| 2000-2049 | Heap, stack, memory |
| 2050-2099 | Ordinal and long integer arithmetic |
| 2100-2149 | Real and double real arithmetic |
| 2150-2199 | Structures, sets, and strings |
| 2200-2399 | Reserved |
| 2400-2449 | Unused |
| 2450-2499 | Other internal errors |
| 2500-2999 | Reserved |

The source program context is optional; it is controlled by metacommands. If the $ENTRY metacommand is on the program context consists of

- the source file name of the compiland containing the error
- the name of the routine in which the error occurred (program, unit, module, procedure, or function)

- the line number of the routine in the listing file
- the page number of the routine in the listing file

If the $LINE metacommand is also on, the line number of the statement containing the error is also given. Setting $LINE also sets $ENTRY.

**Machine Error Context**

By default, run-time routines are compiled with the $RUNTIME metacommand set. This generates special calls for each run-time routine at the entry and exit points so that, for any error that occurs in a run-time routine, the location of that error is in the user program. The entry call, BRTEQQ, saves the context (frame pointer, stack pointer, and program counter) at the point where the run-time routine is called by the user program. The exit call restores the context. The run-time entry helper, BRTEQQ, uses the run-time values shown in Table 19-4.

**Table 19-4. Run-Time Values in BRTEQQ.**

| Value | Description |
|---|---|
| RESEQQ | Stack pointer |
| REFEQQ | Frame pointer |
| REPEQQ | Program counter offset |
| RECEQQ | Program counter segment |

The first thing that BRTEQQ does is examine RESEQQ. If this value is not zero, the current run-time routine was called from another run-time routine and the error context has already been set, so it just returns. If RESEQQ is zero, however, the error context must be saved. The caller's stack pointer is determined from the current frame pointer and stored in RESEQQ. The address of the caller's saved frame pointer and

return address (program counter) in the frame is determined. Then the caller's frame pointer is saved in REFEQQ. The caller's program counter (for example, BRTEQQ's caller's return address) is saved: the offset in REPEQQ and the segment (if any) in RECEQQ.

The run-time exit helper, ERTEQQ, has no parameters. It determines the caller's stack pointer (again, from the frame pointer) and compares it against RESEQQ. If these values are equal, the original run-time routine called by your program is returning, so RESEQQ is set back to zero.

EMSEQQ uses RESEQQ, REFEQQ, REPEQQ and RECEQQ to display the machine error context.

### Source Error Context

Giving the source error context involves extra overhead, since source location data must be included in the object code in some form. This is done with calls that set the current source context as it occurs. These calls can also be used to break program execution as part of the debug process. The overhead of source location data, especially line number calls, can be significant. Routine entry and exit calls, while requiring more overhead individually, are much less frequent, so the overall overhead is less.

The procedure entry call to ENTEQQ passes two VAR parameters: the first is a LSTRING containing the source file name; the second is a record that contains the following:

- the line number of the procedure (a WORD)
- the page number of the procedure (a WORD)
- the procedure or function identifier (an LSTRING)

The file name is that of the compiland source (the main source file name, not the names of any $INCLUDE files.) If one name is given in an INTERFACE and another in a USES clause, the USES identifier is used. The line and page are those designated by the procedure header.

Entry and exit calls are generated for the main program, unit initialization, and module initialization, in which case the identifier is the program, unit, or module name, respectively.

The procedure exit call to EXTEQQ does not pass any parameters. It pops the current source routine context off a stack maintained in the heap.

The line number call to LNTEQQ passes a line number as a value parameter. The current line number is kept in the PUBLIC variable CLNEQQ. Since the current routine is always available (because $LINE implies $ENTRY), the compiland source file name and the name of the routine containing the line are available along with the line number. Line number calls are generated just before the code in the first statement on a source line. The statement can, of course, be part of a larger statement. The $LINE+ metacommand should be placed at least a couple of symbols before the start of the first statement intended for a line number call. ($LINE- also takes effect early.)

Most of the error handling routines are in modules ERRE and PASE. The source error context entry points ENTEQQ, EXTEQQ, and LNTEQQ are in the debug module DEBE.

## AVOIDING THE USE OF RUN-TIME ROUTINES

You may wish to write programs with Pascal that are specifically designed to use a minimum amount of memory. To do so, you should not use Pascal features that call run-time routines in your source code, and should avoid linking your program to the run-time library.

Use of the file input/output, real numbers, and sets all involve the run-time library routines. Units involve use of the run-time library, although use of modules does not. Use of the $DEBUG metacommand also brings in the run time. Section 14, "Available Procedures and Functions," indicates which procedures and functions are implemented through the run-time library.

The Pascal run-time modules linked with a Pascal program may occupy from 36.5 to 70K bytes of memory. Out of that, 4 to 5.5K bytes are taken by the run-time data. Run-time data, the user's data, the stack, and the short heap all share one memory segment (64K bytes). For more information, see your current Pascal Release Notice.

You can suppress linking the run-time library by explicitly specifying the module [sys]<sys>PasMin.obj in the object module line of the Linker command form. In this case, your program must provide the run-time support that is normally provided by the Pascal run time. This includes file and memory management and also all the run-time services that use the file and memory management (for instance, the 8087 emulator). If you do link in PasMin.Obj, you can enter either "Yes" or "No" for [DS Allocation?].

A useful technique when avoiding the run-time library support is to enter "none" as the last parameter for the [Libraries] field of the Linker command form. This ensures that [Sys]<Sys>Pascal.Lib is not linked to your program and the run time cannot be accessed. Any calls made to the run time then appear as unresolved external references. (See the subsection "Linking a Pascal Program" in Section 18, "Using the Pascal Compiler," for an example of how to complete the [Libraries] field.)

**EXAMPLES**

Each sample program below performs the same function. The first program does not use the run-time routines.

**Example 1: Min.Pas**

```
{$debug-}

Program TypeFile_NoRunTime;

    {This program does not use any elements of the
    Pascal run-time system.  ByteStreams are used
    in place of Pascal I/O and CTOS parameter
    management is used instead of Pascal parameter
    management.  Also the metacommand '$debug-' is
    included to turn off the run-time error
    checking.}

Const
        modeRead=#6d72;
        modeWrite=#6d77;

Type
        pbType  =ads of word;
        ppType  =ads of pbType;
        sdType  =record
                   pb   [00]:pbType;
                   cb   [04]:word;
                   end;
        pSdType =ads of sdType;


Function RgParam (
    iParam,
    jParam :word;
    pSdRet :pSdType ) :word; extern;

Function OpenByteStream (
    pBswa       :pbType;
    pbFileSpec  :pbType;
    cbFileSpec  :word;
    pbPassword  :pbType;
    cbPassword  :word;
    mode        :word;
    pbBuffer    :pbType;
    cbBuffer    :word ) :word; extern;

Function ReadByte (
    pBswa :pbType;
    pByte :pbType ) :word; extern;
```

```
Function WriteByte (
    pBswa :pbType;
    b     :byte ) :word; extern;

Function CloseByteStream (
    pBswa :pbType ) :word; extern;

Procedure CheckErc (
    erc :word ) ; extern;


Var     [public]
        erc,
        cbRet   :word;
        bswa    :array [1..130] of byte;
        bsBuffer:array [1..1024] of byte;
        b       :byte;

Var     [extern]
        bsVid   :byte;   {open video bytestream
                from CTOS.Lib}


Procedure Init[public];
    var sd :sdType;
    begin
    CheckErc (RgParam (1,0, ads sd));
    {get 1st Executive paramameter, the file to be
    typed, and open it}
    CheckErc (OpenByteStream (ads bswa,
                sd.pb,
                sd.cb,
                ads ' ',
                0,
                modeRead,
                ads bsBuffer,
                1024));
    end;

Procedure TypeFile[public];
    begin
    While true do
        begin
        erc := ReadByte (ads bswa, ads b);
        if erc<>0 then break;{end of file}
        CheckErc (WriteByte (ads bsVid, b));
        end;
    CheckErc (CloseByteStream (ads bswa));
    end;
```

```
begin   {program start}
    Init;
    TypeFile;
end.
```

**Example 2: Max.Pas**

```
Program                          TypeFile_UsingRunTime
(Input,OutPut,lsFileSpec);

    {This program types the file specified by the
    first parameter of a command form
    ('lsFileSpec'), to the Video}


Var     [public]
        inputFile,
        outputFile :file of byte;
        lsFileSpec :lstring(91);
        b          :byte;

Procedure Init[public];
    begin
    {the Pascal initialization run time loads
    lsFileSpec, see "program" statement with the
    first field of the Executive command form}

    inputFile.trap := true;{trap I/O errors}
    Assign (inputFile, lsFileSpec);
    Reset (inputFile);
    Assign (outputFile, '[vid]');
    Rewrite (outputfile);
    end;


Procedure TypeFile[public];
    begin
    While true do
        begin
        Read (inputFile, b);
        if inputFile.errs <> Ø then break;{end of
               file}
        Write (outputFile,b);
        end;
    end;

begin{program start}
    Init;
    TypeFile;
end.
```

## APPENDIX A: COMPILER ERROR MESSAGES

This section lists error messages generated by the Pascal compiler. For operating system status messages and error codes see the Status Codes Manual.

### ERRORS DETECTED BY THE FRONT END (PARSER/SEMANTIC ANALYZER

Front end error and warning messages include a number as well as a message, and most contain a row of dashes and an arrow to the location of the error. The front end recovers from most errors. However a few such errors are called panic errors, in which case the front end only lists the rest of the program. Panic errors also give the message:

Compiler Cannot Continue!

and occur in the following conditions:

- Error count set by $ERRORS exceeded.
- End of file occurs when not expected.
- Identifier scopes too deeply nested.
- Cannot find PROGRAM, MODULE, or IMPLEMENTATION keyword.
- Cannot find PROGRAM, MODULE, or IMPLEMENTATION identifier.

The word "Warning" before a message indicates the intermediate code files produced by the front end are correct, and the condition is not severe or is just considered "unsafe." Other messages indicate true errors; writing to the intermediate files stops, and these files are discarded when the front end is finished.

The error message "Compiler" refers to an internal consistency check which failed; no matter what source program is compiled, there should be no way to get one of these messages. The comment in this list refers to the compiler routine containing the call.

**FRONT END ERROR LIST**

| Decimal Value | Meaning |
|---|---|
| 101 | Invalid Line Number |
| | Line number is above 32767; there are too many lines in the source file. |
| 102 | Line Too Long Truncated |
| | Source lines are currently limited to 142 characters. |
| 103 | Identifier Too Long Truncated |
| | Any identifier longer than the maximum is truncated. |
| 104 | Number Too Long Truncated |
| | Numeric constants are limited to the identifier length. |
| 105 | End of String Not Found |
| | The line ended before the closing quote was found. |
| 106 | Assumed String |
| | A double quote (") or an accent mark (`) is assumed to enclose a string; use a single quote (') instead. |
| 107 | Unexpected End of File |
| | End of file appears in a number, or metacommand, etc. [while scanning]. |
| 108 | Metacommand Expected Command Ignored |
| | A $ at the start of a comment is not followed by an identifier. |
| 109 | Unknown Metacommand Ignored |
| | A metacommand identifier was unknown or invalid in this version. |

| Decimal Value | Meaning |
|---|---|
| 110 | Constant Identifier Unknown or Invalid Assumed Zero<br><br>A metacommand is set to a constant identifier (as in $DEBUG: A) and the identifier is unknown or not constant of the right type. |
| 111 | [Unassigned] |
| 112 | Invalid Numeric Constant Assumed Zero<br><br>A metacommand is set to a numeric constant (as in $DEBUG: 1) and the constant has the wrong format or is out of range. |
| 113 | Invalid Meta Value Assumed Zero<br><br>A metacommand is set to neither a constant or identifier. |
| 114 | Invalid Metacommand<br><br>One of +, -, or : is expected following a metacommand. |
| 115 | Wrong Type Value for Metacommand Skipped<br><br>The metacommand expects a string but an integer is given, or vice versa. |
| 116 | Meta Value Out of Range Skipped<br><br>o The $LINESIZE integer value was below 16 or above 160.<br><br>o The $REAL:N integer value was not 4 or 8.<br><br>o The $INTEGER:N integer value was not 2. |
| 117 | File Identifier Too Long Skipped<br><br>The $INCLUDE string value for the filename was too long. |

| Decimal Value | Meaning |
|---|---|
| 118 | Too Many File Levels<br><br>There are too many $INCLUDE file nesting levels. |
| 119 | Invalid Initialize Meta<br><br>A $POP metacommand has no corresponding $PUSH metacommand. |
| 120 | CONST Identifier Expected<br><br>A $INCONST metacommand was not followed by an identifier. |
| 121 | Invalid INPUT Number Assumed Zero<br><br>The user input invoked by $INCONST was invalid in some way. |
| 122 | Invalid Metacommand Skipped<br><br>A $IF and its value was not followed by $THEN or $ELSE. |
| 123 | Unexpected Metacommand Skipped<br><br>A $THEN, $ELSE, or $END was found unrelated to a $IF metacommand. |
| 124 | Unexpected Metacommand<br><br>The metacommand was not in a comment; it was processed anyway. |
| 125 | Assumed Hexadecimal<br><br>A # was led without a "16" warning. |
| 126 | Invalid Real Constant<br><br>A type REAL constant was used with a leading or trailing decimal point. |
| 127 | Invalid Character Skipped<br><br>Source file character is not acceptable in program text. |

| Decimal Value | Meaning |
|---|---|
| 128 | Forward Proc Missing<br><br>The procedure or function given in the message was declared FORWARD but not found. [Message occurs in $SYMTAB area.] |
| 129 | Label Not Encountered<br><br>The label given in the message was declared or used in a GOTO, BREAK, or CYCLE but not found. [Message occurs in $SYMTAB area]. |
| 130 | Program Parameter Bad<br><br>The program parameter given in the message was never declared or has the wrong type for READFN. [Message occurs in $SYMTAB area]. |
| 131 | [Unassigned] |
| 132 | [Unassigned] |

NOTE

The following overflow errors can occur in several contexts.

| | |
|---|---|
| 133 | Type Size Overflow<br><br>The data type implies a structure bigger than 32766 bytes. |
| 134 | Constant Memory Overflow<br><br>Constant memory allocation has gone above 65534 bytes. |
| 135 | Static Memory Overflow<br><br>Static memory allocation has gone above 65534 bytes. |

| Decimal Value | Meaning |
|---|---|
| 136 | Stack Memory Overflow<br><br>Stack frame memory allocation has gone above 65534 bytes. |
| 137 | Integer Constant Overflow<br><br>A type INTEGER or other, signed constant expression out of range. |
| 138 | Word Constant Overflow<br><br>A type WORD or other unsigned constant expression is out of range. |
| 139 | Value Not in Range for Record<br><br>Record tag value is not in range of variant, in a structured constant, a long form NEW/DISPOSE/SIZEOF, or other application. |
| 140 | Too Many Compiler Labels<br><br>The compiler needs internal labels; the program is too big. |
| 141 | Compiler [in BOUNDS]<br><br>This refers to an internal consistency check which failed; no matter what source program is compiled, this message should not occur. The compiler is in error, not your source program. The comment in this list refers to the compiler routine containing the call. |
| 142 | Too Many Identifier Levels<br><br>Identifier scope level is over 15. (This is a compiler panic error. See explanation at the front of this section.) |

| Decimal Value | Meaning |
|---|---|
| 143 | Compiler [in DECLEVL]<br><br>This refers to an internal consistency check which failed; no matter what source program is compiled, this message should not occur. The compiler is in error, not your source program. The comment in this list refers to the compiler routine containing the call. |
| 144 | Compiler [in LOOKUP7; often a PASKEY file format error]<br><br>If this error occurs, you can rename the file [Sys]<Sys>Paskey to another name (thus, saving it) and try to recompile your program. However, this error refers to an internal consistency check that failed; no matter what source program is compiled, this message should not occur. |
| 145 | Identifier Already Declared<br><br>An identifier can only be declared once in a given scope level. |
| 146 | Unexpected End of File<br><br>End of file in a statement, declaration, etc. [while parsing]. |

NOTE

The following common substitution mistakes get their own special messages, and are corrected with just a warning.

| | |
|---|---|
| 147 | : Assumed = |
| 148 | = Assumed : |
| 149 | := Assumed = |

| Decimal Value | Meaning |
|---|---|
| 150 | = Assumed := |
| 151 | [ Assumed ( |
| 152 | ( Assumed [ |
| 153 | ) Assumed ] |
| 154 | ] Assumed ) |
| 155 | ; Assumed , |
| 156 | , Assumed ; |
| 157 to 161 | [Unassigned] |

NOTE

If a particular symbol is expected in the source but not found, it may be inserted with one of the following messages.

| | |
|---|---|
| 162 | Insert Symbol<br>[this message should not occur; it is a minor compiler error] |
| 163 | Insert , |
| 164 | Insert ; |
| 165 | Insert = |
| 166 | Insert := |
| 167 | Insert OF |
| 168 | Insert ] |
| 169 | Insert ) |
| 170 | Insert [ |
| 171 | Insert ( |

| Decimal Value | Meaning |
|---|---|
| 172 | Insert DO |
| 173 | Insert : |
| 174 | Insert . |
| 175 | Insert .. |
| 176 | Insert END |
| 177 | Insert TO |
| 178 | Insert THEN |
| 179 | Insert * |
| 180 to 184 | [Unassigned] |

> NOTE
>
> If a particular symbol is expected in the source but is found after some invalid symbols, the invalid ones are deleted with the following two messages.

185 Invalid Symbol - Begin Skip

186 End Skip

187 End Skip

The previous error message ended with the phrase "Begin Skip"; this message marks the end of skipped source text.

188 Section or Expression Too Long

Compiler limit; try rearranging the program or breaking up long expressions by assigning intermediate values to temporary variables.

| Decimal Value | Meaning |
|---|---|
| 189 | Invalid Set Operator or Function<br><br>These include, for example, MOD operator or ODD function with sets. |
| 190 | Invalid Real Operator or Function<br><br>These include, for example, MOD operator or ODD function with reals. |
| 191 | Invalid Value Type for Operator or Function<br><br>These include, for example, MOD operator or ODD function with enumerated type. |
| 192 | [Unassigned] |
| 193 | [Unassigned] |
| 194 | Type Too Long<br><br>A variable or type with greater than 32766 bytes is used. |
| 195 | Compiler [in SIZEOFT, {B}]<br><br>This refers to an internal consistency check that failed; no matter what source program is compiled, this message should not occur. The compiler is in error, not your source program. The comment in this list refers to the compiler routine containing the call. |
| 196 | Zero Size Value<br><br>Use of the empty record "RECORD END" as if it had a size. |
| 197 | Compiler [in ALLOCAT, {B}]<br><br>This refers to an internal consistency check that failed; no matter what source program is compiled, this message should not occur. The compiler is in error, not your source program. The comment in this list refers to the compiler routine containing the call. |

| Decimal Value | Meaning |
|---|---|
| 198 | Constant Expression Value out of Range<br><br>Check array index, subrange assignment, other subrange check. |
| 199 | Integer Type Not Compatible with Word Type<br><br>A common error that indicates confusing signed and unsigned arithmetic; either change the positive signed value to unsigned with WRD () or change the unsigned value (< MAXINT) to signed with ORD (). |
| 200 | [Unassigned] |
| 201 | Types Not Assignment Compatible<br><br>Check assignment statement or value parameter; see the subsection "Type Compatibility" in Section 4, "Introduction to Data Types." |
| 202 | Types Not Compatible in Expression<br><br>Expression mixes incompatible types; see the subsection "Type Compatibility" in Section 4, "Introduction to Data Types." |
| 203 | Not Array - Begin Skip<br><br>A variable followed by a left bracket (or parenthesis) is not an array. |
| 204 | Invalid Ordinal Expression Assumed Integer Zero<br><br>The expression has the wrong type or a type that is not ordinal. |
| 205 | Invalid Use of PACKED Components<br><br>A component of a PACKED structure has no address (it may not be on a byte boundary); it cannot be passed by reference. |

| Decimal Value | Meaning |
|---|---|
| 206 | Not Record Field Ignored<br><br>A variable followed by a dot is not a record, address, or file. |
| 207 | Invalid Field<br><br>A record variable and dot are not followed by a valid field. |
| 208 | File Dereference Considered Harmful<br><br>When the address of a file buffer variable is calculated, the special actions normally done with buffer variables, that is, lazy evaluation (for textfiles) or concurrency (for binary files), cannot be done; the buffer variable at this address may not be valid. (See Section 7, "Files," and Section 15, "File-Oriented Procedures and Functions.") |
| 209 | Cannot Dereference Value<br><br>A variable followed by a caret is not a pointer, address, or file. |
| 210 | Invalid Segment Dereference<br><br>A variable resides at a 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<br><br>A constant ordinal expression was expected. |
| 212 | [Unassigned] |
| 213 | [Unassigned] |
| 214 | Out of Range for Set - 255 Assumed<br><br>An element of a set constant must have an ordinal value <= 255. |

| Decimal Value | Meaning |
|---|---|
| 215 | Type Too Long or Contains File - Begin Skip<br><br>A structured constant must have 255 or fewer bytes; also, it cannot be or contain a file type or an LSTRING type. |
| 216 | Extra Array Components Ignored<br><br>An array constant has too many components for the array type. |
| 217 | Extra Record Components Ignored<br><br>A record constant has too many components for the record type. |
| 218 | Constant Value Expected Zero Assumed<br><br>A value in a structured constant is not constant. |
| 219 | [Unassigned] |
| 220 | Compiler [in STRCONS]<br><br>This refers to an internal consistency check which failed; no matter what source program is compiled, this message should not occur. The compiler is in error, not your source program. The comment in this list refers to the compiler routine containing the call. |
| 221 | Components Expected for Type<br><br>A structured constant needs more components for its type. |
| 222 | Overflow - 255 Components in String Constant<br><br>A string constant must have 255 or fewer bytes. |
| 223 | Use NULL<br><br>The predeclared constant NULL must be used instead of two quotes. |

| Decimal Value | Meaning |
|---|---|
| 224 | Cannot Assign with Supertype LSTRING<br><br>A super array LSTRING cannot be source or the target of assignment. |
| 225 | String Expression Not Constant<br><br>String concatenation with the asterisk only applies to constants. |
| 226 | String Expected Character - 255 Assumed<br><br>Somehow a string constant had no characters, perhaps using NULL. |
| 227 | Invalid Address of Function<br><br>Assignment or other address reference to the function value is not in the scope of the function. This error also occurs when RESULT is used outside the scope of the function. |
| 228 | Cannot Assign to Variable<br><br>Assignment to READONLY, CONST, or FOR control variable. |
| 229 | [Unassigned] |
| 230 | Unknown Identifier Assumed Integer - Begin Skip<br><br>Unknown identifier, for which the address is needed. |
| 231 | VAR Parameter or WITH Record Assumed Integer - Begin Skip<br><br>Invalid identifier, for which the address is needed. |
| 232 | Cannot Assign to Type<br><br>The target of assignment is a file or otherwise cannot be assigned. |

| Decimal Value | Meaning |
|---|---|
| 233 | Invalid Procedure or Function Parameter - Begin Skip |

An error in the use of an intrinsic procedure or function, such as the following:

- The first parameter to NEW or DISPOSE is not a pointer variable.
- The long form of a NEW/DISPOSE/SIZEOF record tag value was not found.
- The long form of a NEW/DISPOSE/SIZEOF super array, has too many bounds.
- The long form of a NEW/DISPOSE/SIZEOF super array, does not have enough bounds.
- A NEW or SIZEOF super array was not given bounds.
- ORD or WRD was performed on a value that is not of an ordinal type.
- LOWER or UPPER was performed on an invalid value or type.
- PACK or UNPACK was performed on a super array, array of files.
- The first parameter to RETYPE is not a type identifier.
- A RESULT parameter is not a function identifier.
- A CODEBYTE parameter value is greater than 255.
- An intrinsic is used which is not available in this version.
- ORD or WRD of an INTEGER4 value out of range.
- A HIWORD or LOWORD parameter is not ordinal or INTEGER4.

| Decimal Value | Meaning |
|---|---|
| 234 | Type Invalid Assumed Integer<br><br>o A parameter to READ, WRITE, ENCODE, or DECODE is not of type INTEGER, WORD, REAL, BOOLEAN, enumerated, or pointer.<br><br>o A parameter to READ and WRITE is not of type CHAR, STRING, or LSTRING.<br><br>o A parameter to READFN is not of type FILE.<br><br>o A program parameter does not have a "readable" type; in this case the error occurs at the BEGIN keyword for the main program. |
| 235 | Assumed File INPUT<br><br>The first READFN parameter is not a file, so INPUT is assumed. |
| 236 | Not File Assumed Text File<br><br>The first parameter to READ or WRITE (or READLN or WRITELN) was assumed to be the file but this assumption was not correct; please give INPUT or OUTPUT explicitly to avoid this message. |
| 237 | Assumed INPUT<br><br>INPUT was not given as a program parameter. |
| 238 | Assumed OUTPUT<br><br>OUTPUT was not given as a program parameter. |
| 239 | LSTRING Expected<br><br>The target of a READSET, ENCODE, or DECODE must be an LSTRING. |
| 240 | [Unassigned] |

| Decimal Value | Meaning |
|---|---|
| 241 | Invalid Segment Variable<br><br>The variable resides at a segmented address, but a default segment address is needed. You may need to make local copy of the variable. |
| 242 | File Parameter Expected - Begin Skip<br><br>READSET expects a textfile parameter. |
| 243 | Character Set Expected<br><br>READSET expects a SET OF CHAR parameter. |
| 244 | Unexpected Parameter - Begin Skip<br><br>EOF, EOLN, and PAGE do not take more than one parameter. |
| 245 | Not Text File<br><br>EOLN, PAGE, READLN and WRITELN only apply to textfiles. |
| 246 | [Unassigned] |
| 247 | Invalid Function<br><br>Use of the intrinsic function WRD is invalid. |
| 248 | Size Not Identical<br><br>The warning is given in RETYPE; it may or may not work as intended. |
| 249 | Procedural Type Parameter List Not Compatible<br><br>The parameter lists for formal and actual procedural parameters are not compatible. The number of parameters is different: the function result type or parameter type is different: or the attributes are wrong. |

| Decimal Value | Meaning |
|---|---|
| 250 | Cannot Use Procedure with Attribute<br><br>You cannot call an INTERRUPT procedure, directly or indirectly. |
| 251 | Unexpected Parameter - Begin Skip<br><br>The procedure or function has no parameters, but a left parenthesis was found. |
| 252 | Cannot Use Procedure or Function as Parameter<br><br>An intrinsic procedure or function cannot be passed as parameter. |
| 253 | Parameter Not Procedure or Function - Begin Skip<br><br>A procedural parameter was expected; you need a procedure or function here. |
| 254 | Supertype Array Parameter Not Compatible<br><br>Actual parameter is not same or derived super type as formal. |
| 255 | Compiler [in ACTUALS]<br><br>This refers to an internal consistency check which failed; no matter what source program is compiled, this message should not occur. The compiler is in error, not your source program. The comment in this list refers to the compiler routine containing the call. |
| 256 | VAR or CONST Parameter Types Not Identical<br><br>Actual and formal reference parameter types must be identical. |
| 257 | Parameter List Size Wrong - Begin Skip<br><br>Too few or too many parameters were used; skips only if too many. |

| Decimal Value | Meaning |
|---|---|
| 258 | Invalid Procedural Parameter to EXTERN<br><br>The actual procedure or function is invoked with intrasegment calls, and so cannot be passed to an external code segment. Give the PUBLIC attribute to the procedure or function to fix this. |
| 259 | Invalid Set Constant for Type<br><br>The set is not constant, the base types are not identical, or the constant is too big. |
| 260 | Unknown Identifier in Expression Assumed Zero<br><br>The identifier is undefined (or misspelled) in an expression. |
| 261 | Identifier Wrong in Expression Assumed Zero<br><br>A general identifier error in an expression has occurred; for example, file type id. |
| 262 | Assumed Parameter Index or Field - Begin Skip<br><br>After error 260 or 261, anything in parentheses or square brackets, or a dot followed by an identifier, is skipped. |
| 263 | [Unassigned] |
| 264 | [Unassigned] |
| 265 | Invalid Numeric Constant Assumed Zero<br><br>A decode error in an assumed INTEGER (or WORD) literal constant. |
| 266 | [Unassigned] |

| Decimal Value | Meaning |
|---|---|
| 267 | Invalid Real Numeric Constant<br><br>A decode error in an assumed type REAL literal constant. |
| 268 | Cannot Begin Expression Skipped<br><br>A symbol cannot start an expression, so it has been deleted. |
| 269 | Cannot Begin Expression Assumed Zero<br><br>A symbol cannot start an expression, so zero has been inserted. |
| 270 | Constant Overflow<br><br>DIV or MOD by the constant zero (INTEGER or WORD). |
| 271 | Word Constant Overflow<br><br>Unary minus, on a WORD operand (try NOT word + 1). |
| 272 | Word Constant Overflow<br><br>WORD constant minus a WORD constant gives a negative result. |
| 273 | [Unassigned] |
| 274 | [Unassigned] |
| 275 | Invalid Range<br><br>The lower bound of a subrange is greater than upper bound (e.g., 2..1). |
| 276 | CASE Constant Expected<br><br>A constant value is expected for a CASE statement or record variant. |
| 277 | Value Already in Use<br><br>In a CASE statement or record variant, a value has already been assigned (as in CASE 1..3: XXX; 2: YYY; END). |

| Decimal Value | Meaning |
|---|---|
| 278 | Invalid Symbol<br><br>".." was used in a CASE or record variant. |
| 279 | Label Expected<br><br>In a BREAK, CYCLE, or GOTO statement, or starting a statement, or in a LABEL section, the expected label was not found. |
| 280 | Invalid Integer Label<br><br>Nondecimal notation (e.g., 8#77, etc.) is not allowed in labels. |
| 281 | Label Assumed Declared<br><br>This label did not appear in the LABEL section. |
| 282 | [Unassigned] |
| 283 | Expression Not Boolean Type<br><br>The expression following IF, WHILE, or UNTIL must be BOOLEAN. |
| 284 | Skip to End of Statement<br><br>An unexpected ELSE or UNTIL clause was skipped. |
| 285 | Compiler [in STATEMT {B}]<br><br>This refers to an internal consistency check which failed; no matter what source program is compiled, this message should not occur. The compiler is in error, not your source program. The comment in this list refers to the compiler routine containing the call. |
| 286 | ; Ignored<br><br>A semicolon before ELSE is always in error, and is skipped. |

| Decimal Value | Meaning |
|---|---|
| 287 | [Unassigned] |
| 288 | : Skipped<br><br>A colon after OTHERWISE is always in error, and is skipped. |
| 289 | Variable Expected For FOR Statement - Begin Skip<br><br>A variable identifier must come after FOR. |
| 290 | [Unassigned] |
| 291 | FOR Variable Not Ordinal or Static or Declared in Procedure<br><br>The FOR statement control variable must not be<br><br>o type REAL, INTEGER4, or other non-ordinal type<br><br>o the component of an array, record, or file type<br><br>o the referent of a pointer type or address type<br><br>o in the stack or heap, unless locally declared<br><br>o nonlocally declared, unless in static memory<br><br>o a reference parameter (VAR or VARS parameter)<br><br>o a variable with a segmented ORIGIN attribute |
| 292 | Skip to :=<br><br>In a FOR statement, the assignment is expected here. |

| Decimal Value | Meaning |
|---|---|
| 293 | GOTO Invalid<br><br>The GOTO or label here involves an invalid GOTO statement. |
| 294 | GOTO Considered Harmful<br><br>The $GOTOCK metacommand is on, and here is a GOTO. |
| 295 | [Unassigned] |
| 296 | Label Not Loop Label<br><br>The BREAK or CYCLE label is not before a FOR, WHILE, or REPEAT. |
| 297 | Not in Loop<br><br>The BREAK or CYCLE statement is not in a FOR, WHILE, or REPEAT. |
| 298 | Record Expected - Begin Skip<br><br>A WITH statement expects a record variable. |
| 299 | [Unassigned] |
| 300 | Label Already in Use Previous Use Ignored<br><br>This label has already appeared in front of a statement. |
| 301 | Invalid Use of Procedure or Function Parameter<br><br>A procedure parameter was used as a function, or vice versa. |
| 302 | [Unassigned] |
| 303 | Unknown Identifier Skip Statement<br><br>The starting statement identifier is undefined (or misspelled). |

| Decimal Value | Meaning |
|---|---|
| 304 | Invalid Identifier Skip Statement<br><br>A general identifier error starts a statement; for example, file type id. |
| 305 | Statement Not Expected<br><br>A MODULE or uninitialized IMPLEMENTATION with a main BEGIN..END. |
| 306 | Function Assignment Not Found<br><br>Somewhere in the function's body its value must be assigned. |
| 307 | Unexpected END Skipped<br><br>An END was unexpected; perhaps a missing BEGIN, CASE, or RECORD. |
| 308 | Compiler [in CONTEXT {B}]<br><br>This refers to an internal consistency check which failed; no matter what source program is compiled, this message should not occur. The compiler is in error, not your source program. The comment in this list refers to the compiler routine containing the call. |
| 309 | Attribute Invalid<br><br>An attribute valid only for procedures and functions was given for variable or vice versa, or an invalid attribute mix such as PUBLIC and EXTERN was used. |
| 310 | Attribute Expected<br><br>A left bracket indicates attributes, but this is not an attribute. |
| 311 | Skip to Identifier<br><br>This symbol was skipped to get to the identifier which follows. |

| Decimal Value | Meaning |
|---|---|
| 312 | Identifier Expected<br><br>A list of identifiers is expected, but this is not an identifier. |
| 313 | [Unassigned] |
| 314 | Identifier Expected Skip to ;<br><br>A new identifier to be declared was expected but not found. |
| 315 | Type Unknown or Invalid Assumed Integer - Begin Skip<br><br>Parameter or function return type not identifier, undeclared, or value parameter or function return with file or super array. |
| 316 | Identifier Expected<br><br>No identifier appears after a PROCEDURE or FUNCTION in a parameter list. |
| 317 | [Unassigned] |
| 318 | Compiler internal error. |
| 319 | Compiler internal error. |
| 320 | Previous Forward Skip Parameter List<br><br>The parameter list and function return type are not repeated when a forward (or interface) procedure or function is defined. |
| 321 | Not EXTERN<br><br>A procedure or function with the ORIGIN attribute must be EXTERN. |
| 322 | Invalid Attribute with Function or Parameter<br><br>An INTERRUPT procedure cannot have parameters or be a function. |

| Decimal Value | Meaning |
|---|---|
| 323 | Invalid Attribute in Procedure or Function<br><br>A nested procedure or function cannot have attributes or be EXTERN. |
| 324 | Compiler internal error. |
| 325 | Already Forward<br><br>FORWARD cannot be used twice for the same procedure or function. |
| 326 | Identifier Expected for Procedure or Function<br><br>The keywords PROCEDURE or FUNCTION must be followed by an identifier. |
| 327 | Invalid Symbol Skipped<br><br>FORWARD or EXTERN directives are never used in interfaces. |
| 328 | EXTERN Invalid with Attribute<br><br>An EXTERN procedure cannot have the PUBLIC attribute. |
| 329 | Ordinal Type Identifier Expected Integer Assumed - Begin Skip<br><br>An ordinal type identifier is expected for a record tag type. |
| 330 | Contains File Cannot Initialize<br><br>A file in a record variant, while allowed, is considered unsafe and is not initialized automatically with the usual NEWFQQ call. |
| 331 | Type Identifier Expected Assumed Integer<br><br>This error occurs when an ordinal type identifier is expected. |

| Decimal Value | Meaning |
|---|---|
| 332 | Invalid Type<br><br>Declaring the WORD type. |
| 333 | Not Supertype Assumed String<br><br>This looks like a super array type designator but type identifier is not a super array type so STRING super array type is assumed. |
| 334 | Type Expected Integer Assumed<br><br>This is a general message; a type clause or type identifier is expected. |
| 335 | Out of Range 255 for LSTRING<br><br>An LSTRING designator cannot have an upper bound over 255. |
| 336 | Cannot Use Supertype Use Designator<br><br>Super array type must be reference parameter or pointer referent. |
| 337 | Supertype Designator Not Found<br><br>All upper bounds must be given in a super array designator. |
| 338 | Contains File Cannot Initialize<br><br>A super array of a file type, while allowed, is considered unsafe and is not initialized automatically with the usual NEWFQQ call. |
| 339 | Supertype Not Array Skip to ;<br><br>An Integer is assumed. The keyword SUPER is always followed by ARRAY in a type clause. |
| 340 | Invalid Set Range Integer Ø to 255<br><br>The base type of a set must be within the subrange Ø..255. |

| Decimal Value | Meaning |
|---|---|
| 341 | File Contains File<br><br>A file type cannot contain a file type, directly or indirectly. |
| 342 | PACKED Identifier Invalid Ignored<br><br>The PACKED keyword must be followed by one of ARRAY, RECORD, SET, or FILE; it cannot be followed by a type identifier. |
| 343 | Unexpected PACKED<br><br>The PACKED keyword only applies to structured types. (See above.) |
| 344 | [Unassigned] |
| 345 | Skip to ;<br><br>A semicolon is expected at the end of a declaration (not at end of line). |
| 346 | Insert ;<br><br>Semicolon expected at end of declaration (at end of line). |
| 347 | Cannot Use Value Section with ROM Memory<br><br>Setting $ROM on prevents the use of a VALUE section. |
| 348 | UNIT Procedure or Function Invalid EXTERN<br><br>In an IMPLEMENTATION, any interface procedures and functions not implemented must be declared EXTERN at the beginning of the IMPLEMENTATION, but this EXTERN occurs later. |
| 349 | [Unassigned] |

| Decimal Value | Meaning |
|---|---|
| 350 | Not Array - Begin Skip<br><br>A variable in a VALUE section followed by square bracket not array. |
| 351 | Not Record - Begin Skip<br><br>A variable in VALUE section followed by a dot is not a record type. |
| 352 | Invalid Field<br><br>In the VALUE section an identifier assumed to be a field is not in the record. |
| 353 | Constant Value Expected<br><br>In the VALUE section a variable can only be initialized to a constant. |
| 354 | Not Assignment Operator Skip to ;<br><br>The assignment operator was not found in a VALUE section. |
| 355 | Cannot Initialize Identifier Skip to ;<br><br>A symbol in the VALUE section is not a variable declared at this level in fixed (STATIC) memory, or has the ORIGIN or EXTERN attribute. |
| 356 | Cannot Use Value Section<br><br>Put the VALUE section in the IMPLEMENTATION, not the INTERFACE. |
| 357 | Unknown Forward Pointer Type Assumed Integer<br><br>The identifier for the referent of a reference type declared earlier in this TYPE (or VAR) section was never declared itself. |

| Decimal Value | Meaning |
|---|---|
| 358 | Pointer Type Assumed Forward<br><br>In this TYPE section, a pointer or address type occurred in which the referent type was already declared in an enclosing scope, but the identifier for the referent type was declared again later in the same TYPE section. For example: TYPE A=WORD; PROCEDURE B; TYPE C=^A; A=REAL;Message says the forward type is used in this case (such as, REAL). |
| 359 | Cannot Use Label Section<br><br>Put a LABEL section in the IMPLEMENTATION, not the INTERFACE. |
| 360 | Forward Pointer to Supertype<br><br>The referent of a reference type declared in this TYPE section is a super array type; the supertype declaration must come earlier. |
| 361 | Constant Expression Expected Zero Assumed<br><br>In a CONST section, the expression is not constant. |
| 362 | Attribute Invalid<br><br>In a VAR section, PUBLIC or ORIGIN with are used with EXTERN, or ORIGIN in attribute brackets after the VAR keyword. |
| 363 | [Unassigned] |
| 364 | Contains File Initialize Module<br><br>File variables must be initialized. Thus, when a file variable is declared in a module the module must be called (as a parameterless procedure) to initialize these files. |

| Decimal Value | Meaning |
|---|---|
| 365 | Origin Variable Contains File Cannot Initialize<br><br>File variables must be initialized, but ORIGIN variables are never initialized, so the user must initialize this file. |
| 366 | UNIT Identifier Expected Skip to ;<br><br>USES was not followed by the identifier of a unit. |
| 367 | Initialize Module to Initialize UNIT<br><br>A USES clause triggers a unit initialization call, but to invoke this call the module must be called as a procedure. |
| 368 | Identifier List Too Long - Extra Assumed Integer<br><br>In a USES clause with a list of identifiers, more identifiers were found in the list than are constituents of the interface. |
| 369 | End of UNIT Identifier - List Ignored<br><br>In a USES clause with a list of identifiers, fewer identifiers were found in the list than are constituents of the interface. |
| 370 | [Unassigned] |
| 371 | UNIT Identifier Expected<br><br>After the phrase INTERFACE; a UNIT identifier was not found. |
| 372 | Compiler error<br><br>This error occurs when the keyword UNIT is missing in an interface. |

| Decimal Value | Meaning |
|---|---|
| 373 | Identifier in UNIT List Not Declared<br><br>One of the identifiers in the interface UNIT list was not declared in the body of the interface. |
| 374 | Program Identifier Expected<br><br>No identifier appears after the PROGRAM or MODULE keyword. (This is a compiler panic error. See explanation at the front of this section.) |
| 375 | UNIT Identifier Expected<br><br>No unit identifier after IMPLEMENTATION OF. (This is a compiler panic error. See explanation at the front of this section.) |
| 376 | Program Not Found<br><br>PROGRAM, MODULE, or IMPLEMENTATION OF keywords not found (panic). Can occur if source file is not a Pascal compiland. |
| 377 | File End Expected Skip to End<br><br>The assumed end of the compiland was processed, but there is more. |
| 378 | Program Not Found<br><br>The main body of a PROGRAM or initialized IMPLEMENTATION, or the final END of a MODULE or other IMPLEMENTATION, was not found. |

**ERRORS DETECTED BY THE BACK END (OPTIMIZER/CODE GENERATOR)**

The following program errors are detected by the back end:

- Attempt to divide by zero. For example,

  A DIV Ø.

- Overflow during integer constant folding. For example,

  MAXINT+A+MAXINT.

- Expression too complex or too many internal labels.

  (Try breaking up the expression by using assignments to temporary variables.)

The optimizer and code generator perform a large amount of internal consistency checking. When one of these checks encounters an unexpected condition, the result is an internal error generated by the module where the inconsistency was discovered.

Such errors should generally not occur. When they do occur, we request that they be reported promptly. Since it may be difficult to analyze such reports unless they include the complete source code involved, please include the complete source code in a machine readable form.

The format of an optimizer error message is as shown below:

```
*** Internal Error  <error number>
 Near Line <source line number>
 Contact Technical Support
```

where <error number> is an internal error number and <source line number> is the last source line number seen by the optimizer. The error may not have occurred exactly at this line, but it is likely to be within a few lines following this line. The <source line number> corresponds to the line numbers on the listing generated by the front end.

**Module OPTIM (Status Numbers Ø to 99)**

1 Bad ICode file format (PRSDEC1Ø).

2 Bad symbols file format; cannot find function return variable (READ_SYMTAB).

3 Multiple symbols file entries for symbol that is not a procedure or function (READ_SYMTAB).

4 Forward reference to an Icod4 number (XLATE).

5 ICode reference to a missing symbol (XLATE_SYM).

6 Duplicate ICode numbers in same block (ENTER_XLATE).

7 Invalid or unexpected operand for ADDR ICode (PHASE1).

8 Invalid addressing mode for ADDR ICode (PHASE1).

9 Invalid or unexpected operand for DRRR ICode (PHASE1).

1Ø Invalid or unexpected operand for DRFR ICode (PHASE1).

11 Invalid symbol type for UPPR ICode (PHASE1).

12 Invalid addressing mode for ASMS/ASVS (PHASE1).

13 Bad tree format; assignment target tree does not have a SYMR node as its leftmost lead (DEL_TARGET).

14 Unknown ICode value (SUREX).

15 Bad statement list returned from SPLITTREE (OPTIM - main program).

16 Bad statement list returned from PHASE1 (OPTIM).

17 Bad statement list returned from CHECK_LENGTH (OPTIM).

18 Bad statement list returned from PHASE2 (OPTIM).

19 Bad statement list returned from PHASE3 (OPTIM).

2Ø Bad statement list returned from MD_XFURM (OPTIM).

21 Bad statement list returned from SUREX (OPTIM).

**Module GEN6 (Status Numbers 100 to 199)**

100 Static nesting level < 0 (NESTLEV).

101 Invalid or unexpected operand for OFFR ICode (MD_XFORM).

102 Invalid flag values for CONR (MD_XFORM).

103 Invalid or unexpected operand for UPPR ICode (MD_XFORM).

104 Invalid symbol type for UPPR operand (MD_XFORM).

105 Too many levels of indirection for UPPR operand (MD_XFORM).

106 Invalid addressing mode for VALP ICode (MD_XFORM).

107 Invalid or unexpected operand for LVAP ICode (MD_XFORM).

108 Multiple definition of an internal label (GENDONE).

109 Cannot load long constant value with a length > 4 (CASELONR).

110 Invalid offset value for OFSR ICode (CASEOFSR).

111 Register table entry or use count for OFSR ICode is bad (CASEOFSR).

112 Invalid nesting for procedure/function call (CALLPF).

113 Invalid function return length (CASECALP).

114 Bad use count for SFRT ICode operands (CASESFRT).

115 Symbol type is invalid, must be a variable (CLASS).

116 Operand use count is already 0 (COUNTUSE).

117 User label must begin a basic block (DEF_ULAB).

118 Duplicate definition of user label (DEF_ULAB).

119 Address flag missing for LONR ICode (EMITIMM).

120 Address flag missing for SYMR ICode (EMITIMM).

121 Variable must be static (EMITIMM).

122 Symbol type must be variable (EMITIMM).

123 Invalid ICode type (EMITIMM).

124 Cannot save a multibyte value (EMPTYREG).

125 Invalid register contents (EMPTYREG).

126 Symbol type must be variable (GENREF).

127 Missing address flag for long constant reference (GENREF).

128 Invalid ICode type (GENREF).

129 Value must be in an index register (GENREF).

130 Value must be in an index register (GENREFI).

131 No registers available for allocation (GETREG).

132 Register BX already in use (IMBXES).

133 Register must be SI or DI (INDREF).

134 Symbol must be variable (INDREF).

135 Missing address bit for long constant reference (LOADR).

136 Symbol must be a variable (LOADR).

137 Invalid ICode type (LOADR).

138 Symbol type must be a label (LONGGOTO).

139 Register residence flags do not match register table contents (MOVER).

140 Value must be in some register (REGN).

141 Invalid operand register specified by template (REGSPEC).

142 Operand's register residence flag does not match the specified register (REGSPEC).

143 Operand must be in a register (X_BINOP).

144 Unexpected opcode value (X_BINOP).

145 Contents of BX do not match operand (X_CHKBXES).

146 Invalid ICode operator (X_CMPI).

147 Invalid ICode operand (X_CMPI).

148 Invalid variable kind (must be static) or address bit missing (X_CMPI).

149 Invalid ICode operator; must have two operands (X_COMOPR).

150 Desired register already in use (WANTREG).

151 Desired register already in use (X_DONE).

152 Index must already be in a register (X_DONEA).

153 Invalid register contents (X_DONEA).

154 Symbol must be a variable (X_DONEA).

155 Invalid ICode operand (X_DONEA).

156 Invalid condition code for IF template (X_IFCOND).

157 Invalid condition code for IFOPR template (X_IFOCOND).

158 Register BX contents are wrong (X_INREGS).

159 Source register is empty (X_MOVREG).

160 Register residence flag for operand is bad (X_MOVREG).

161 Invalid register designated; cannot access high half of register (X_SELFH).

163 Invalid ICode for assignment target (X_STOR).

164 Invalid ICode operator; must have two operands (X_REVOPR).

165 Invalid opcode value (X_UNIOP).

166 Invalid register specification (X_XCHG).

167 Cannot exchange registers containing part of a multiregister value (X_XCHG).

168 Register residence flag does not match register table contents (X_XCHG)

169 Register residence flag does not match register table contents (X_XCHG).

170 Register table contents do not match their associated register residence flags (INTERPRET).

171 Operand must be a CONR node (INTERPRET).

172 No match for this operand class in the templates for this ICode (SCANCLASS).

173 Register BX is already in use (INTERPRET).

174 Use count was not decremented properly (INTERPRET).

175 Use count was not decremented properly (INTERPRET).

176 Error in template processing (INTERPRET).

177 Invalid register specification; cannot access high/low half of the register (INTERPRET).

178 Invalid or unexpected template operator (INTERPRET).

179 Invalid length for OFFR ICode; must be length 1, 2, or 4 (GEN_SUBTREE).

180 Symbol table entry for RTPP ICode does not match the current procedure/function (GEN_SUBTREE).

181 Symbol table entry for RTPP ICode must be a procedure or function (GEN_SUBTREE).

182 Invalid or unexpected ICode value (GEN_SUBTREE).

**Module SUBR (Status Numbers 200 to 299)**

200 Value too large to convert to WORD type, BOOT compiler only (WRDTOINT).

201 Missing address bit for assignment target (TARGCHECK).

202 Invalid ICode for assignment target (TARGCHECK).

203 Unexpected opcode value, BOOT compiler only (GET_OPCFLAGS).

204 Invalid opcode flag value (GETTYP).

205 Invalid opcode flag value (GETTYP).

**Module FOLD (Status Numbers 300 to 399)**

300 Invalid operand count, must have two operands (FOLD_CONS).

301 Invalid constant values for operands to the NOTB ICode (FOLD_CONS).

**Module CHKLEN (Status Numbers 400 to 499)**

400 Operand length cannot be 0 (CHECKLEN).

401 Length of operands must match if both are greater than 0 (CHECKLEN).

402 Operand length must be -1, 1, or 2 (CHECKLEN).

403 Operand length must be -1, 1, or 2 (MUST1OR2).

404 New length must be 1 or 2 (COERCE).

405 Assignment target must be variable or function (TARG_LEN).

406 Invalid ICode for assignment target (TARG_LEN).

407 Invalid symbol type for SYMR ICode (CHECK_LENGTH).

408 Assignment target length must be 4 for AS48 (CHECK_LENGTH).

409 Invalid addressing for VAXP operand (CHECK_LENGTH).

410 Unexpected ICode value (CHECK_LENGTH).

**Module CTL6 (Status Numbers 500 to 599)**

500 Code generator-computed code size does not match the computed code size.

501 Invalid class override, CS_DTYP record (BINPS).

502 Invalid symbol type, CS_SYM record (BINPS2).

503 Internal label reference to an undefined label, CS_CJMP record (BINPS2).

504 Internal label reference to an undefined label, CS_ILAB record (BINPS2).

505 Internal label location does not match current location counter, CS_DILB record (BINPS2).

506 User label reference to an undefined label, CS_ULAB record (BINPS2).

507 User label location does not match current location counter, CS_DULB record (BINPS2).

508 P-code procedure/function entry address does not match current location counter, CS_PFBEG/CS PROB record (BINPS2).

509 Procedure/function entry address does not match current location counter, CS_PFBEG/CS PROB record (BINPS2).

510 Unknown binary interpass file record type (BINPS2).

**Module DUMP86 (Status numbers 600 to 699)**

600 Unexpected interpass record type (GETBYTE).

601 Unexpected end of data (GETDATA).

602 Invalid data size (GETDATA).

603 Invalid data size (GETDATA).

604 Unexpected end of data (GETDISP).

605 Unexpected interpass record type (GETDISP).

606 Unexpected end of data (GETLABEL).

607 Invalid label type, must be short label (GETLABEL).

608 Unexpected interpass record type (GETLABEL).

609 Invalid opcode (WRITEOP).

610 Invalid opcode, no PUSH CS opcode exists (PUSHPOPSEG).

611 Cannot do sign extension on operands for logical operators AND, OR, XOR (BINARYOPS).

612 Invalid mode value (LOADPTR).

613 Invalid opcode value (SHIFTOPS).

614 Unused opcode (GROUPC).

615
to
626 Unused opcode (DUMP86)

**Module DUMP (Status Numbers 700 to 799)**

700 Invalid opcode value (OPNAME).

701 Unknown working value (DMP1ID).

702 Unexpected symbol type (DMP1ID).

703 Invalid operator mode value (WRIMOD).

704 Unexpected ICode value (DMPNOD).

705 Unexpected interpass record type (DMPBREC).

## RUN-TIME ERROR MESSAGES

Errors detected at run time are either file system errors or other program exceptions. File system errors are described first.

### FILE SYSTEM ERRORS

File system error codes range from 1000 to 1999 and are based on the ERRC field of the file control block.

```
? Error: <error type> error in file <file name>
  Error Code <error code>,  System status <status code>
  PC=<program counter>,FP=<frame pointer>,SP=<stack pointer>
```

852-013

File system errors are reported in the following format:

If <error code> is in the range 1000 to 1099, then the error was detected by the CTOS operating system and <status code> is a CTOS status code. See the Status Codes Manual for interpretation of status codes.

If <error code> is in the range 1100 to 1999, then the error was detected by the Pascal file system. These error codes are explained below:

| Decimal Value | Meaning |
|---|---|
| 1100 | ASSIGN or READFN of file name to open file. |
| 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). |

| Decimal Value | Meaning |
|---|---|
| 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. |

| Decimal Value | Meaning |
|---|---|
| 1124 | Encode error (field width > 255) in textfile WRITE REAL. |
| 1125 | Encode error (field width > 255) in textfile WRITE WORD. |
| 1126 | Decode error in textfile READ INTEGER4. |
| 1127 | Encode error in textfile WRITE INTEGER4. |

The <error type> field of the file system error report is based on the ERRS field of the file control block. Error types are described below:

Ø (no error).

1 Hard data. Hard data error.

2 Device name. Invalid device or volume name.

3 Operation. Invalid operation: GET if EOF, RESET a printer, etc.

4 File system. File system internal error.

5 Device offline. Device or volume no longer available.

6 Lost file. File no longer available.

7 File name. Invalid syntax, name too long, etc.

8 Device full. Disk full, directory full, etc.

9 Unknown device. Device or volume not found.

1Ø File not found.

11 Protected file.

12 File in use.

13 File not open.

14 Data format. Data format, decode, or range error.

15 Line too long. Buffer overflow.

## OTHER RUN-TIME ERRORS

Nonfile system error codes range from 2000 to 2999. In some cases, metacommands control whether errors are checked. In other cases, they are always checked. The metacommand controlling a check, if any, is given in the list below.

### 2000 to 2049 Memory Errors

Since the stack and the heap grow toward each other, these errors are all related; for example, a stack overflow can cause a "Heap is Invalid" error if $STACKCK is off and the stack overflows.

| Decimal Value | Meaning |
|---|---|
| 2000 | Stack Overflow<br><br>While calling a procedure or function, the stack ran out of memory. Checked if $STACKCK+ and in some other cases. |
| 2001 | No Room in Heap<br><br>Not enough room is available in the heap for a new variable. This error is always detected. |
| 2002 | Heap Is Invalid<br><br>While allocating memory in the heap for a new variable, an error in the heap structure was found. This error is always detected. |
| 2003 | Heap Allocator Interrupted<br><br>An interrupt procedure was invoked that interrupted NEW and called NEW again. The heap allocator modifies the heap; thus it is a critical section. |
| 2004 | Allocation Internal Error<br><br>An unexpected error return occurred while requesting additional heap space from the operating system. Contact technical support. |

| Decimal Value | Meaning |
|---|---|
| 2031 | Nil Pointer Reference |
| | DISPOSE or $NILCK+ found a pointer with a NIL value. |
| 2032 | Uninitialized Pointer |
| | DISPOSE or $NILCK+ found an uninitialized pointer. Pointers are given this value only if $NILCK is on. |
| 2033 | Invalid Pointer Range |
| | DISPOSE or $NILCK+ found a pointer that does not point into the heap or is otherwise invalid. The pointer may have pointed to a DISPOSED block that was removed from the heap. |
| 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 |
| | When the long form of DISPOSE was used, the actual length of the variable did not equal the length based on the tag values given. |

**2050 to 2099 Ordinal Arithmetic**

| Decimal Value | Meaning |
|---|---|
| 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 is checked if $RANGECK+ is used. |

| Decimal Value | Meaning |
|---|---|
| 2051 | Unsigned Divide by Zero |
| | WORD value divided by zero. This is checked if $MATHCK+ is used. |
| 2052 | Signed Divide by Zero |
| | INTEGER value divided by zero. This is checked if $MATHCK+ is used. |
| 2053 | Unsigned Math Overflow |
| | A WORD result occurred outside 0..MAXWORD. This is checked if $MATHCK+ is used. |
| 2054 | Signed Math Overflow |
| | An INTEGER result occurred outside -MAXINT..MAXINT. This is checked if $MATHCK+ is used. |
| 2055 | Unsigned Value Out of Range |
| | Assignment of a value parameter in which the source value is out of range for the target value. The target can 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. These are checked with $RANGECK+. |
| | Another time this error occurs is when an array index is out of bounds and the array has an unsigned index type. This is checked with $INDEXCK+. |
| 2056 | Signed Value Out of Range |
| | This is the same as 2055, but applies to the INTEGER type and its subranges. |

| Decimal Value | Meaning |
|---|---|
| 2057 | Uninitialized 16-Bit Integer Used |

An INTEGER or 16-bit INTEGER subrange variable is used without being assigned first, or such a variable has the invalid value, -32768. This condition is checked with $INITCK+.

| | |
|---|---|
| 2058 | Uninitialized 8-Bit Integer Used |

A SINT or 8-bit INTEGER subrange variable is used without being assigned first, or such a variable has the invalid value -128. This condition is checked with $INITCK+.

**2100 to 2149 Type REAL Arithmetic**

| Decimal Value | Meaning |
|---|---|
| 2100 | REAL Divide by Zero |

A REAL value was divided by zero. This condition is always detected.

| | |
|---|---|
| 2101 | REAL Math Overflow |

A REAL value is too large for representation. This condition is always detected.

| | |
|---|---|
| 2104 | SQRT of Negative Argument |

A square root function is used on an argument < 0. This condition is always detected.

| | |
|---|---|
| 2105 | LN of Non-Positive Argument |

A natural log function is used on an argument <= 0. This condition is always detected.

| Decimal Value | Meaning |
|---|---|
| 2106 | TRUNC/ROUND Argument Range<br><br>Results from converting a REAL outside the range of INTEGER. This condition is always detected. |
| 2131 | Tangent Argument Too Small<br><br>The tangent argument is so small that the result is invalid. This condition is always detected. |
| 2132 | Arcsin or Arccos of REAL > 1.0<br><br>The arcsin or arccos argument is greater than one. This condition is always detected. |
| 2133 | Negative Real Raised to a Real Power<br><br>An invalid argument in exponentiation. This condition is always detected. |
| 2135 | REAL Math Underflow<br><br>The significance of a REAL expression was reduced to zero. |
| 2136 | REAL Indefinite (uninitialized or previous error)<br><br>The REAL value called "indefinite" was encountered; this can occur if $INITCK was on and an uninitialized real variable was used, or if a previous error set a variable to indefinite as part of its masked error response. |

## 2150 to 2199 Structured Type Errors

| Decimal Value | Meaning |
|---|---|
| 2150 | String Too Long in COPYSTR |
| | A COPYSTR intrinsic source string is too large for target string. This condition is always detected. |
| 2151 | LSTRING Too Long in Intrinsic Procedure |
| | A target LSTRING is too small in INSERT, DELETE, CONCAT, or COPYLST intrinsic procedure. This condition is always detected. |
| 2180 | Set Element Greater Than 255 |
| | A value in a constructed set is above maximum. This condition is always detected. |
| 2181 | Set Element Out of Range |
| | A value in a set assignment or set value parameter is too large for the target set. This condition is detected with $RANGECK+. |

### 2200 to 2249 INTEGER4 Arithmetic Errors

| Decimal Value | Meaning |
|---|---|
| 2200 | INTEGER4 Divide by Zero |
| 2201 | INTEGER4 Math Overflow |
| 2234 | INTEGER4 Zero to Negative Power |

### 2250 to 2999 Other Errors

| Decimal Value | Meaning |
|---|---|
| 2450 | Unit Version Number Mismatch |

During unit initialization, the user (the one with the USES clause) and the implementation of an interface were discovered to have been compiled with unequal interface version numbers. This condition is always detected.

# APPENDIX B: COMPARISONS TO THE ISO STANDARD AND OTHER PASCALS

## COMPARISONS TO THE ISO STANDARD

Our version of Pascal generally conforms to the ISO Pascal standard, Level Ø and Level 1, currently being developed by the ANSI/IEEE committee. However, the conformant array mechanism, a method of passing arrays of different bounds as one parameter type, proposed in Level 1, has not been implemented.

The super array type, a feature of our version of Pascal, provides conformant array parameters, as well as dynamic length arrays allocated on the heap.

In general , programs correctly written to the ISO standard (Level Ø) or to the ANSI/IEEE standard should run correctly, without changes, under this Pascal.

The ISO standard defines a large number of error conditions, but allows a particular implementation to handle an error by documenting the fact that the error is not detected. These "errors not detected," and other differences are described below. A program that conforms or tests conformance to the ISO standard and is written with our version of Pascal must have the metacommand $DEBUG on and must not use any **extend level** features.

The following minor extensions to the current ISO/ANSI/IEEE standard are allowed:

- The question mark (?) can substitute for the caret (^).
- The underscore (_) can be used in identifiers.

Because of the way the compiler binds identifiers, the new reserved words added at the extend level cannot be used as identifiers at the standard level. A new directive, EXTERN, and new predeclared functions are standard in this version of Pascal.

The current differences between the standard level of our version of Pascal and the current

ISO/ANSI/IEEE standard are summarized in the following pages.

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

  In some cases, this version does not require a separator between a number and an identifier or keyword, for example, "100mod" is accepted as "100 mod" without error.

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

  This version of 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 detected. Passing other packed components gives the usual error.

- The ISO standard does not include the textfile line-marker character in the set of CHAR values.

  Our version of Pascal permits all 256 8-bit values as CHAR values; the RETURN character, CHR(10), is also the line marker character.

- The ISO standard requires a variant to be given for all possible tag values. This version does not.

- The ISO standard requires that an identifier have only one meaning in any scope.

  Using an identifier and then redeclaring it in the same scope is an error not detected by this compiler. For example, the following,

```
CONST X=Y;  VAR Y:  CHAR;
```

  has two meanings for Y in the same scope. The latest definition for an identifier is generally used by this version of Pascal. There is one ambiguous case: If you declare type FOO in one scope and in an inner scope TYPE P = ^ FOO; FOO = 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.

- The ISO standard requires field width "M" to be greater than zero in WRITE and WRITELN procedures.

  Our version of 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.

- 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.

  This version of Pascal allows assignments to these variables using the actual length of the target variable. The ISO standard error is not detected.

- The ISO standard does not allow the short form of DISPOSE to be used on a structure allocated with the long form of NEW. The ISO standard only permits a variable allocated with the long form of NEW to be released with the long form of DISPOSE, and no tag fields should change between the calls.

  Our version of 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.

- The ISO standard declares that when a "change of variant" occurs (such as when a new tag value is assigned), all the variant fields become undefined.

  This version of Pascal does not set the fields uninitialized when a new tag is assigned and so does not detect use of a variant field with an undefined value.

- The ISO standard does not allow a variable with an active reference (that is, the records of 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).

  Our version of Pascal does not detect these as errors.

- 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.

  This version of 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.

- The ISO standard at Level 1 defines conformant array.

  Our version of 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.

- The ISO standard requires the control variable of a FOR loop to be local to the immediate block. Any assignment to this control variable is an error.

  This version of Pascal allows a nonlocal variable to be used if it is STATIC, so either a local variable or one at the PROGRAM level can be a FOR statement control variable. This version of Pascal also does not detect an assignment to the control variable as an error if assignment occurs in a procedure or function called within the FOR statement.

- The ISO standard requires the CHR argument to be INTEGER.

  This version of Pascal allows CHR to take any ordinal type.

## SUMMARY OF EXTENSION TO THE ISO STANDARD OFFERED BY OUR VERSION OF PASCAL

This outline summarizes the extensions to the ISO standard which are offered by this version of Pascal. Unless otherwise noted, all are at the **extend level.**

### SYNTACTIC AND PRAGMATIC FEATURES

- the metalanguage at the standard level

| | |
|---|---|
| $BRAVE | $PAGEIF |
| $DEBUG | $PAGESIZE |
| $ENTRY | $POP |
| $ERRORS | $PUSH |
| $GOTO | $RANGECK |
| $INCLUDE | $REAL |
| $INCONST | $ROM |
| $INDEXCK | $RUNTIME |
| $INTICK | $SIMPLE |
| $IF $THEN $ELSE $END | $SIZE |
| $INTEGER | $SKIP |
| $LINE | $SPEED |
| $LINESIZE | $STACKCK |
| $LIST | $SUBTITLE |
| $MATHCK | $SYMTAB |
| $MESSAGE | $TITLE |
| $NILCK | $WARN |
| $OCODE | |
| $PAGE | |

- extra listing at the standard level
  - flags for jumps, globals, identifier level, control level, headers, trailers
  - 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

**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 at the **extend 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 at the extend 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 | PUBLIC |
| EXTERNAL | PURE |
| INTERRUPT | READONLY |
| ORIGIN | STATIC |
| PORT | |

**OPERATORS AND INTRINSICS**

- extend level operators:
  - 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 | UPPER |
| DIV | WRD |
| HIBYTE | * |
| HIWORD | + |
| LOBYTE | - |
| LOWER | < |
| LOWORD | <= |
| MOD | <> |
| ORD | = |
| RETYPE | > |
| SIZEOF | >= |

- additional intrinsic functions at the **extend level:**

| | |
|---|---|
| ABORT | LOWER |
| BYLONG | LOWORD |
| BYWORD | MOVEL |
| DECODE | MOVER |
| ENCODE | MOVESL |
| EVAL | MOVESR |
| FILLC | RESULT |
| FILLSC | RETYPE |
| HIBYTE | SIZEOF |
| HIWORD | UPPER |
| LOBYTE | |

- intrinsic functions that operate on strings:
  - for STRING or LSTRING: COPYSTR POSITN SCANEQ SCANNE
  - for LSTRING only: CONCAT INSERT DELETE COPYLST
- Pascal library functions at the standard level:

| | |
|---|---|
| ALLHQQ | MARKAS |
| BEGOQQ | MEMAVL |
| BEGXQQ | PLYUQQ |
| ENDOQQ | PTYUQQ |
| ENDXQQ | RELEAS |
| FREECT | SADDOK |
| GTYUQQ | SMULOK |
| LADDOK | UADDOK |
| LMULOK | UMULOK |
| LOCKED | UNLOCK |

**CONTROL FLOW AND STRUCTURE FEATURES**

- control flow statements: BREAK, CYCLE, and RETURN
- sequential control operators: AND THEN and OR ELSE in IF, WHILE, REPEAT
- extend 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 number, version checking
  - optional rename of constituents
  - guaranteed unique unit initialization
  - optional unit initialization

**EXTEND LEVEL I/O 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
- full FCBFQQ type equivalent to FILE types.

## COMPARISONS WITH OTHER VERSIONS OF PASCAL

At the standard level, our version of 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 this 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 by this version of Pascal.

## IMPLEMENTATIONS OF PASCAL

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

- o interactive I/O

  Our version of 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.

- o string handling

  This version of 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.

- o compiler controls

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

- o maximum set size

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

- 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.

- out of block GOTOs

  Some Pascals do not permit the out-of-block GOTOs that are permitted in by this version.

- heap management

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

- 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.

- assigning file names

  The ASSIGN procedure in this version of Pascal sets an operating system file name for a file. Some other Pascals use a second parameter to RESET and REWRITE for the file name.

- 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 VAR section, using [PUBLIC] in the PROGRAM and [EXTERN] in the MODULE, and $INCLUDE the same variable declarations in each.

- program parameters

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

- procedural parameters

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

## UCSD PASCAL AND OUR VERSION OF PASCAL

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

Our version of Pascal has incorporated many of the UCSD extensions in one form or another. Table B-1 compares UCSD extensions with similar extensions available in this version.

The following notes describe comparative points of interest.

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

- UCSD Pascal allocates pointer variables on the heap with MARK and RELEASE (in this version MARKAS and RELEAS.) Other Pascals normally use NEW and DISPOSE. Both methods of dynamic memory allocation are available with this version.

- Units are the same, with the following exceptions:

  - In this version of Pascal, an INTERFACE must appear first in any compliance using it. Since UCSD Pascal has its own special file system, the name of the unit can be used to find the interface file name in a standard way.

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

- CONCAT is a function in UCSD Pascal; in our version of Pascal, it is a procedure.

- 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 this version, you must include an empty OTHERWISE clause to obtain this effect.

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

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

**Table B-1. Our Pascal and UCSD Pascal.**

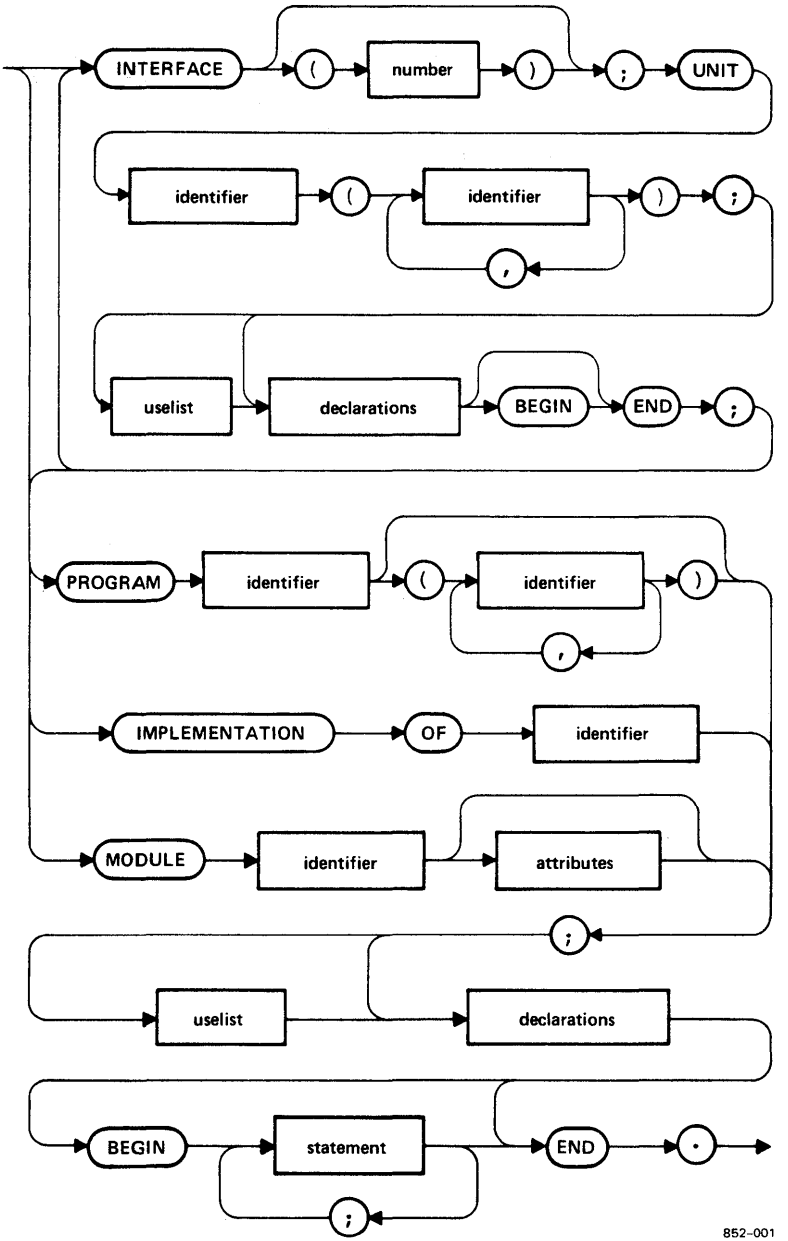
| UCSD Extension | Equivalent |
|---|---|
| ATAN | ARCTAN |
| BLOCKREAD | GETUQQ |
| BLOCKWRITE | PUTUQQ |
| CLOSE | CLOSE |
| CLOSE (F, LOCK) | CLOSE (F) |
| CLOSE (F, PURGE) | DISCARD (F) |
| CONCAT | CONCAT |
| COPY | COPYLST or MOVEL |
| DELETE | DELETE |
| EXIT | RETURN or GOTO |
| FILLCHAR | FILLC 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 | |
| SCAN | SCANEQ and SCANNE |
| SEEK | SEEK |
| SIZEOF | SIZEOF |
| STR | ENCODE |
| STRING [n] | LSTRING (n) |
| UNIT | UNIT |
| Untyped Files | FCBFQQ type |

## APPENDIX C: PASCAL SYNTAX DIAGRAMS

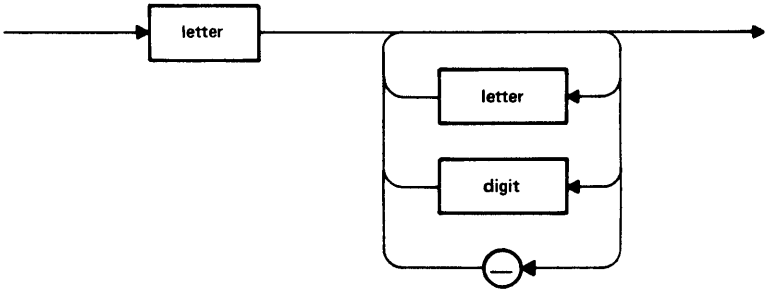
The diagrams on the following pages show the fundamental syntax of the Pascal language. They are arranged in the order that you would be likely to use the elements while writing a program. The meaning of the differently shaped outlines is as follows:

- Ovals — Indicate reserved words or symbols. These must be typed as shown.
- Boxes — Indicate higher-level constructions that usually have syntax diagrams of their own.
- Circles — Indicate punctuation that is required and must be typed as shown.
- Arrows — Help to show the path through the diagram, including any possible looping (that is, repetition of syntax elements.)

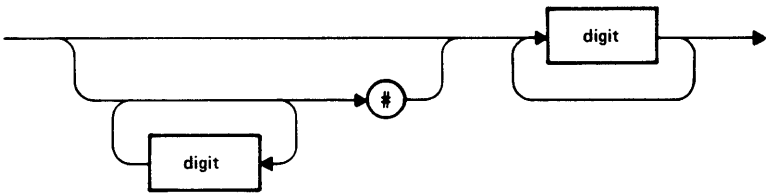
Source File


INTERFACE
(
number
)
;
UNIT
identifier
(
identifier
)
;
,
uselist
declarations
BEGIN
END
;
PROGRAM
identifier
(
identifier
)
,
IMPLEMENTATION
OF
identifier
MODULE
identifier
attributes
;
uselist
declarations
BEGIN
statement
END
.
;
852-001

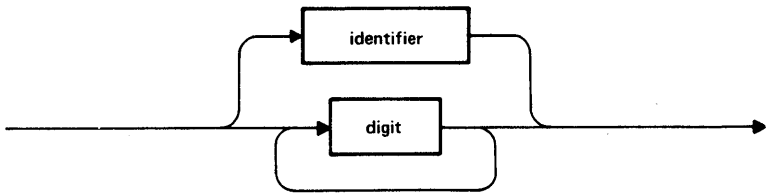
Identifier



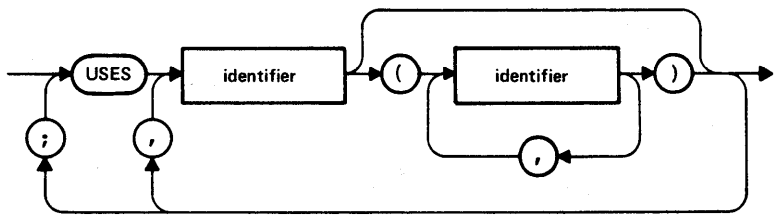

Number



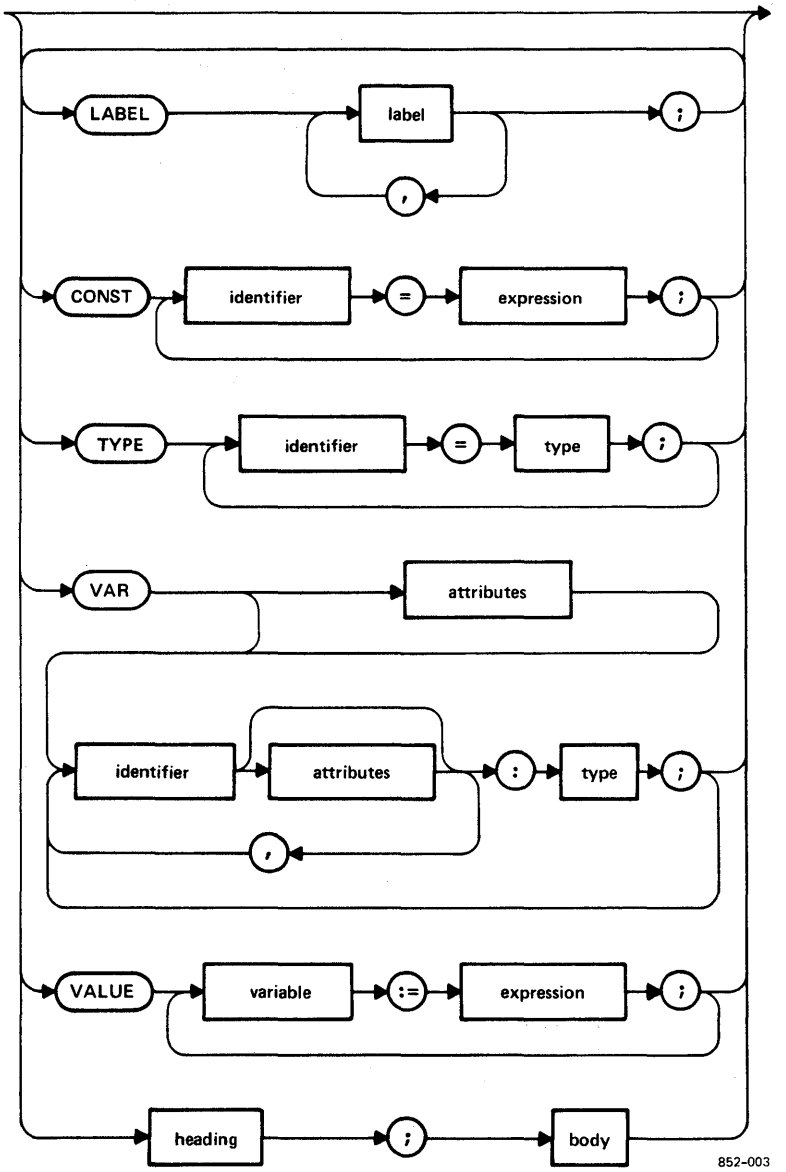

Label



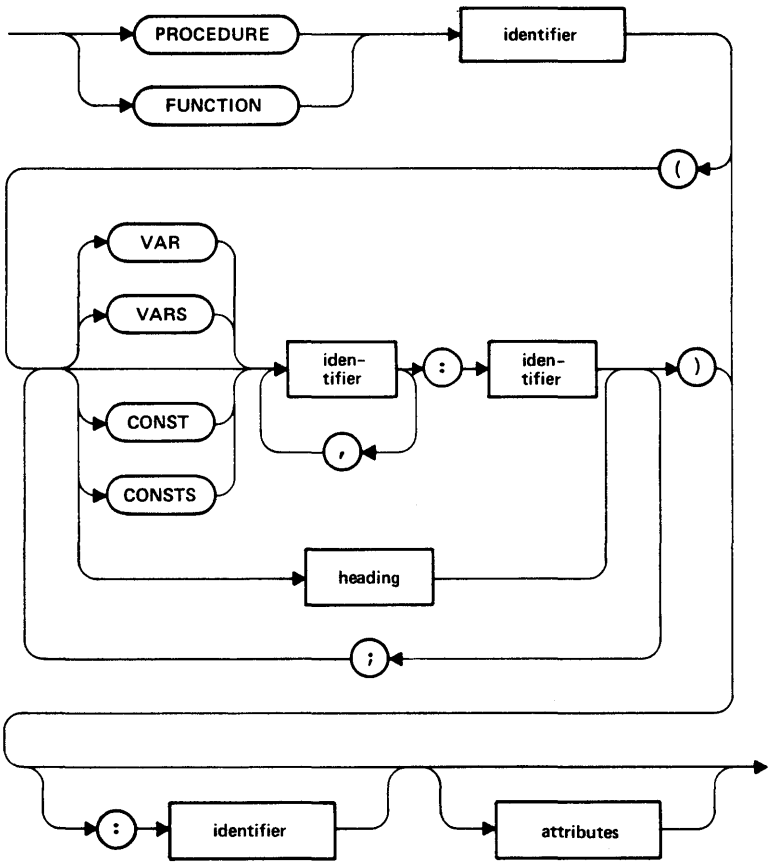

Uselist



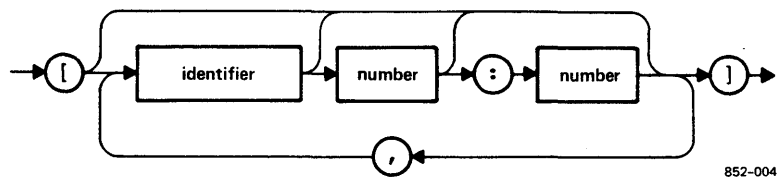

852-002


LABEL
label
;
,
CONST
identifier
=
expression
;
TYPE
identifier
=
type
;
VAR
attributes
identifier
attributes
:
type
;
,
VALUE
variable
:=
expression
;
heading
;
body
852-003

Heading

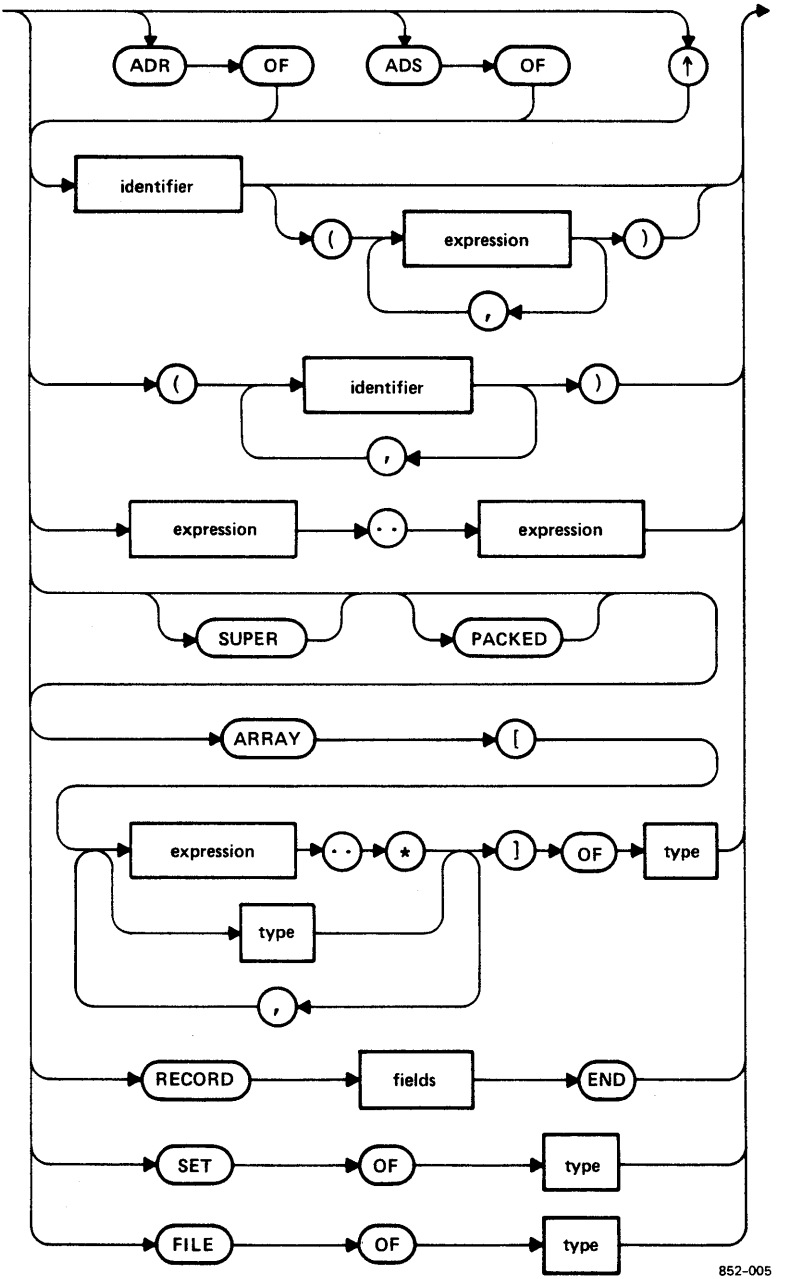



Attributes



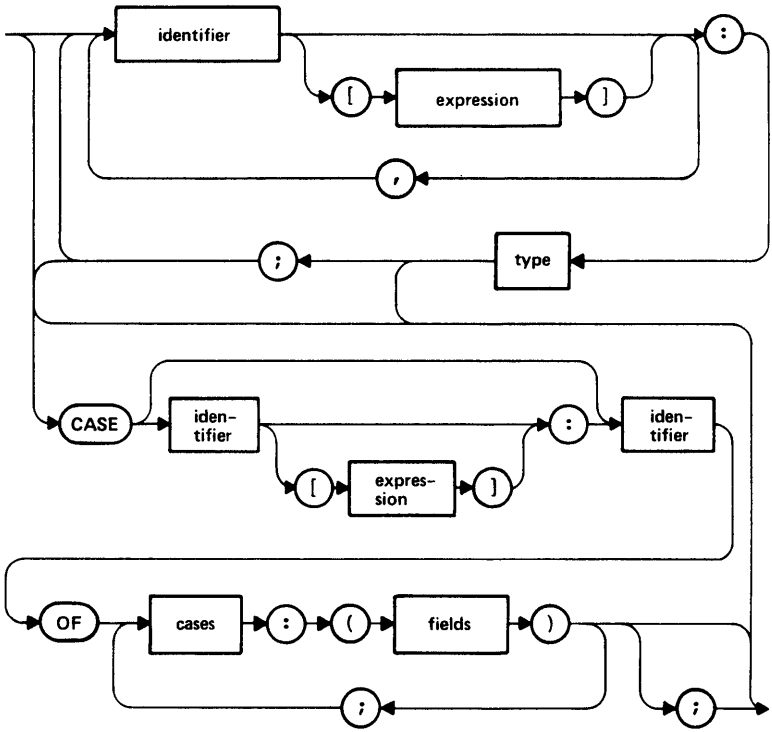

852-004

Type

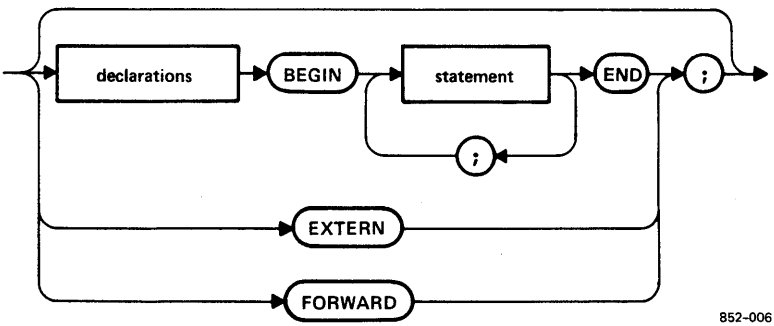

ADR
OF
ADS
OF
↑
identifier
(
expression
)
,
(
identifier
)
,
expression
. .
expression
SUPER
PACKED
ARRAY
[
expression
. .
*
]
OF
type
type
,
RECORD
fields
END
SET
OF
type
FILE
OF
type

852-005

Fields

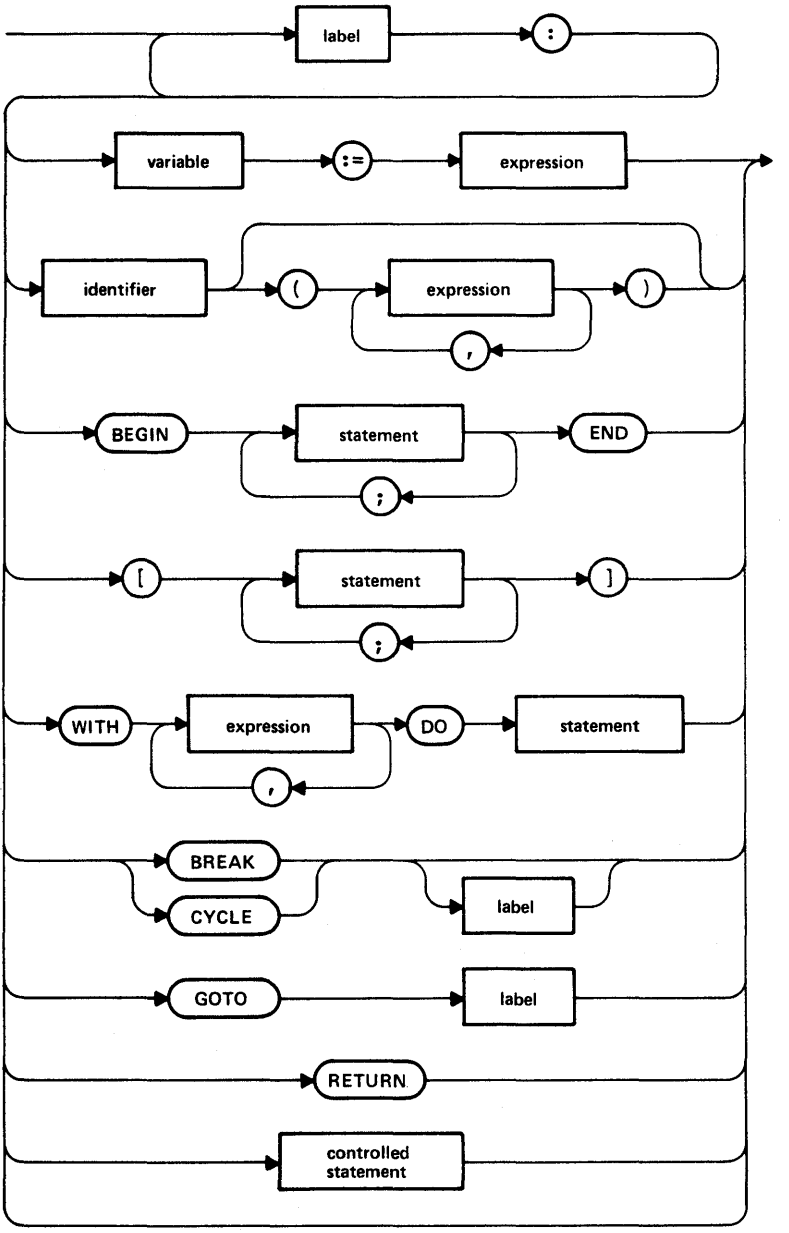



Body



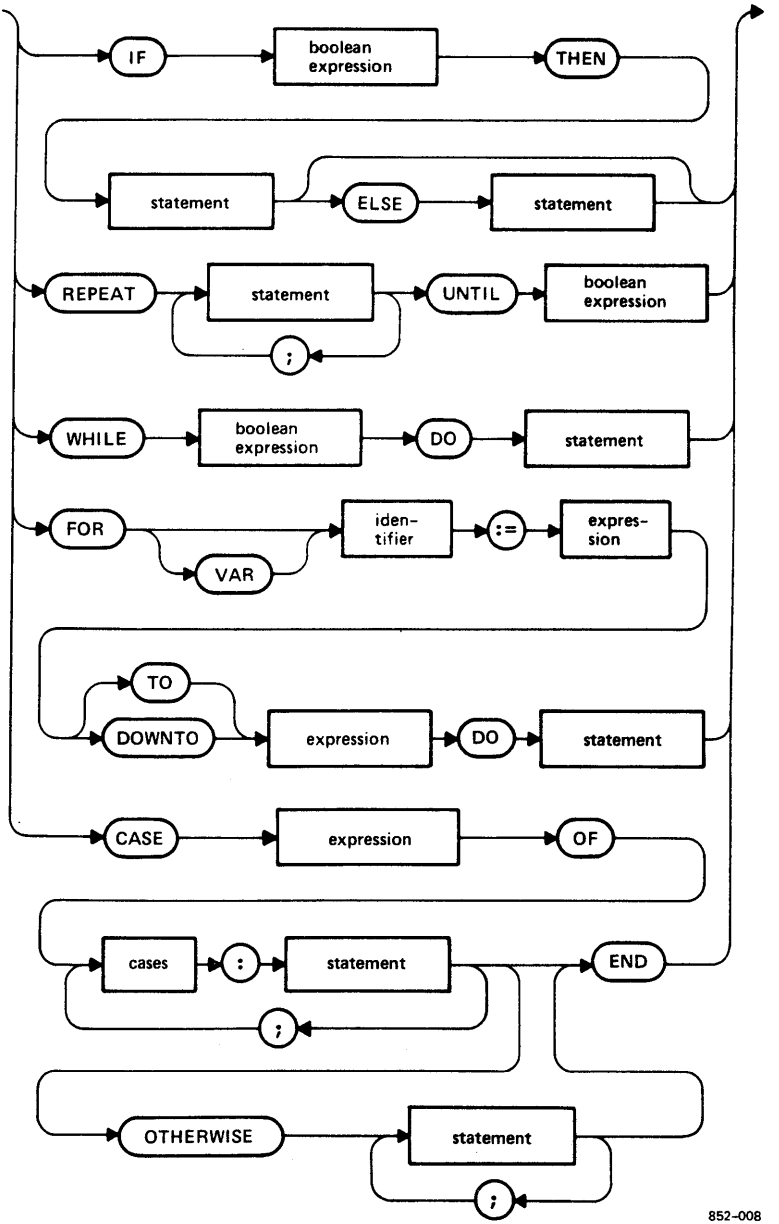

852-006

Statement



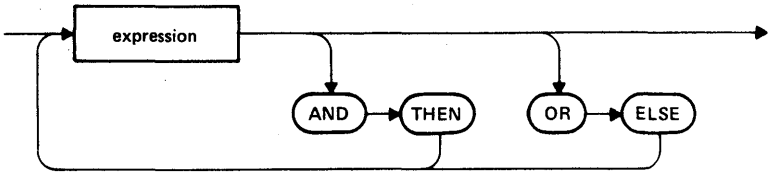

852-007

Controlled Statement

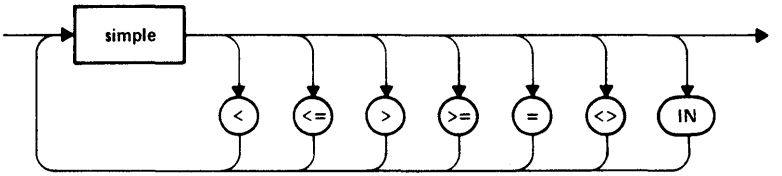



852-008

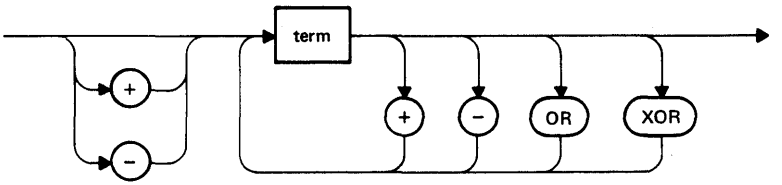
Boolean Expression



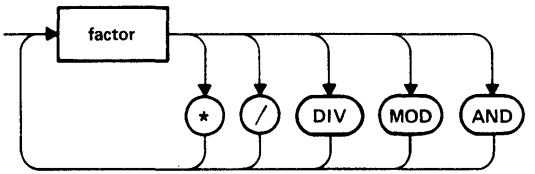

Expression




Simple

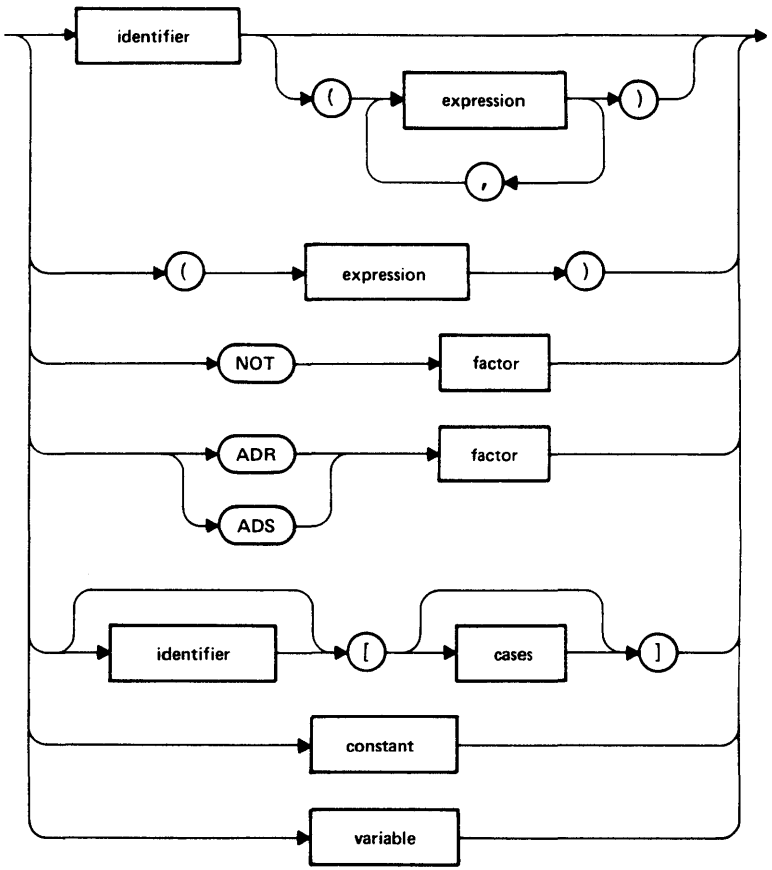



Term

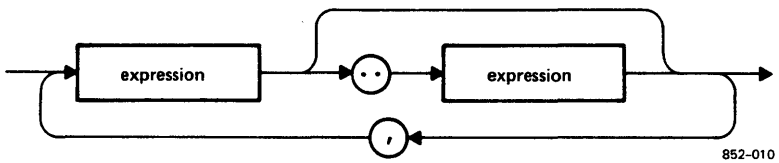



852-009

Factor

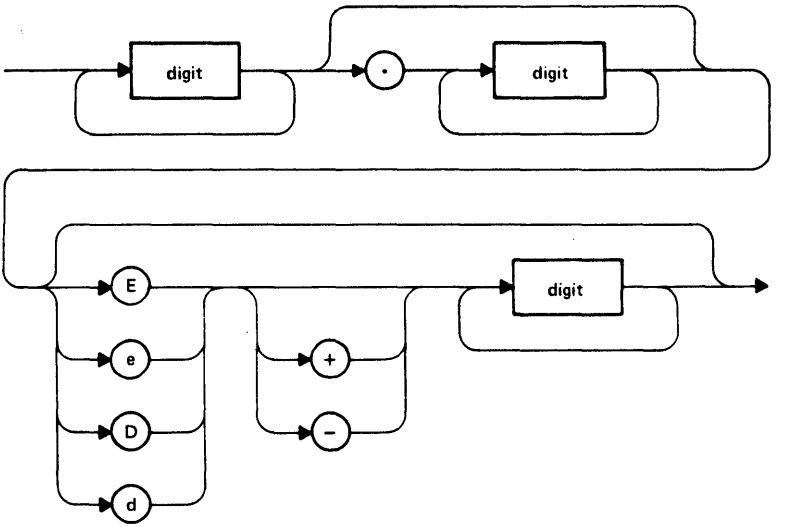



Cases

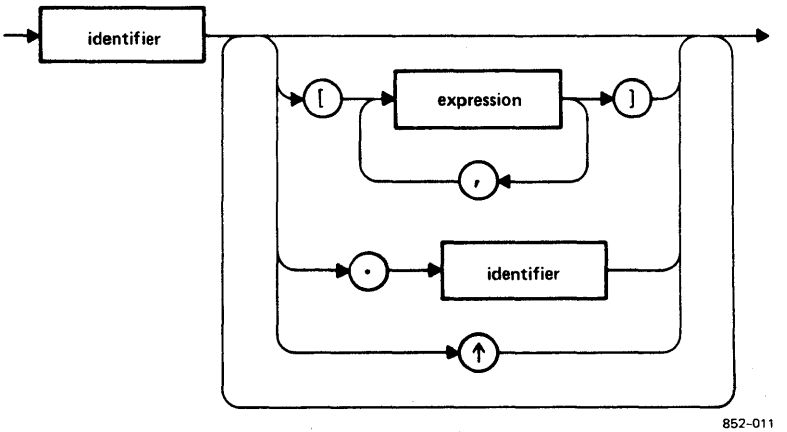



852-010

Real Number

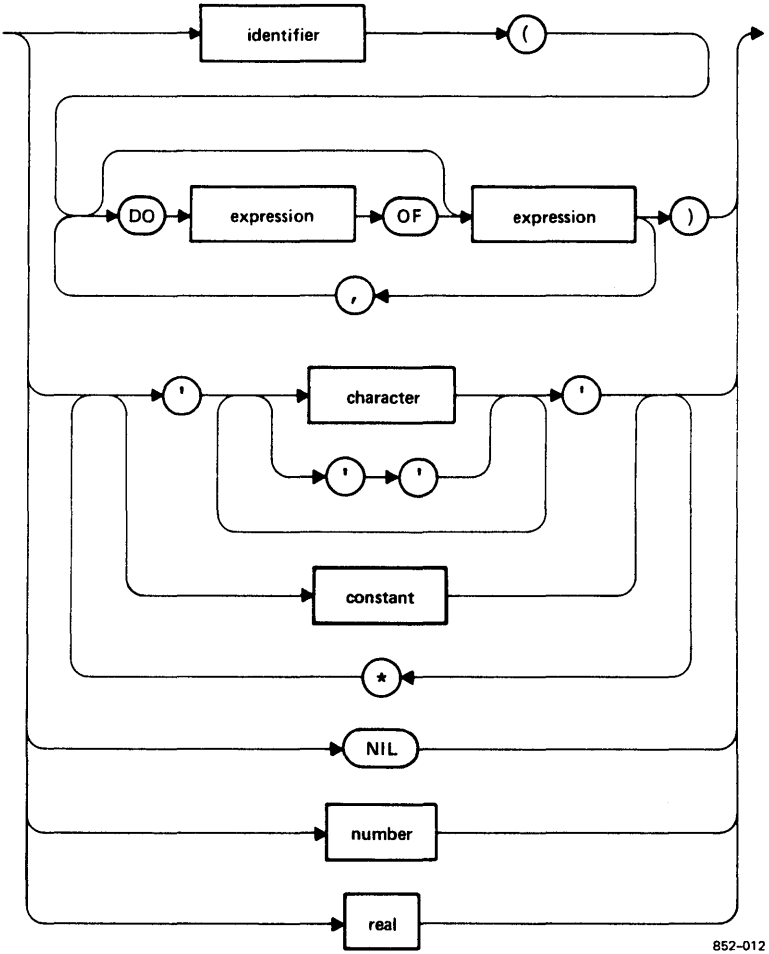



Variable




852-011

Constant




852-012

# APPENDIX D: SUMMARY OF RESERVED WORDS AND PREDECLARED IDENTIFIERS

## RESERVED WORDS

Reserved words at the standard level:

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

Additional reserved words at the **extend level**:

| | |
|---|---|
| ADR | OTHERWISE |
| ADS | RETURN |
| BREAK | UNIT |
| CONSTS | USES |
| CYCLE | VALUE |
| IMPLEMENTATION | VARS |
| INTERFACE | XOR |
| MODULE | |

Names of attributes:

| | |
|---|---|
| EXTERN | PUBLIC |
| EXTERNAL | PURE |
| INTERRUPT | READONLY |
| ORIGIN | STATIC |
| PORT | |

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.

## PREDECLARED IDENTIFIERS

Predeclared identifiers for this version of Pascal are summarized in Tables D-1 and D-2.

**Table D-1. Predeclared Identifiers at the Standard Level.**

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

**Table D-2. Predeclared Identifiers at the Extend Level.**

| | | |
|---|---|---|
| ABORT | FILLSC | READFN |
| ADRMEM | FLOAT4 | READSET |
| ADSMEM | HIBYTE | REAL4 |
| ASSIGN | HIWORD | REAL8 |
| CLOSE | INSERT | RESULT |
| BYLONG | INTEGER1 | RETYPE |
| BYTE | INTEGER2 | ROUND4 |
| BYWORD | INTEGER4 | SCANEQ |
| COPYLST | LOBYTE | SCANNE |
| CONCAT | LOWER | SEEK |
| COPYLST | LOWORD | SEQUENTIAL |
| COPYSTR | LSTRING | SINT |
| DECODE | MAXINT4 | SIZEOF |
| DELETE | MAXWORD | STRING |
| DIRECT | MOVEL | TERMINAL |
| DISCARD | MOVER | TRUNC4 |
| ENCODE | MOVESL | UPPER |
| EVAL | MOVESR | WORD |
| FCBGQQ | NULL | WRD |
| FILEMODES | POSITN | |
| FILLC | | |

## APPENDIX E: CONVERSION TO AND FROM IEEE FORMAT

Pascal releases numbered 8.Ø and higher use IEEE real number format. IEEE format is not compatible with the format used for real numbers in releases of the Pascal compiler numbered less than 8.Ø.

If you need to convert real numbers from one format to the other, you can do so using the following library routines:

- To IEEE Format

  PROCEDURE M21SQQ (VARS RMS, RIEE: REAL4)

- From IEEE Format

  PROCEDURE I2MSQQ (VARS RIEEE, RMS: REAL4)

RMS and RIEEE are real numbers in the old format and IEEE format, respectively.

If you are using the old format, REAL numeric constants must be greater than or equal to 1.ØE-38 and less than 1.ØW+38. For IEEE format, by contrast, REAL numeric constants are kept in double precision and so can range from about 1E-3Ø6 to 1E3Ø6.

# APPENDIX F: USING PASCAL AS A SYSTEMS PROGRAMMING LANGUAGE

Pascal is becoming an increasingly popular language for systems programming in the 8086 environment. The structured approach of the language allows the programmer to easily add assembly language routines wherever necessary for optimum performance. At the same time, the structure of Pascal makes programs that use it easy to read and maintain.

With Pascal, you can access all CTOS operating system services, such as direct (random) access to disk files, interrupt handling, and process creation through Pascal. You can also access DAM, ISAM, Sort/Merge, and the Forms Run Time from Pascal.

Application notes which give examples and detailed information on how to use a variety of CTOS utilities, such as the video and forms, from Pascal are available from technical support.

## MAKING CALLS TO CTOS UTILITIES FROM PASCAL

Pascal allows you to easily access operating system utilities. For instance to make a call to OpenFile simply declare it as a function as below:

```
FUNCTION OpenFile(pfh: FhTypePtr; pbFileSpec:
         StringPtr; CbFileSpec: WORD;
         pbPassword: StringPtr; cbPassword:
         WORD; mode: WORD): ErcType: EXTERN:
```

Since such utilities are used frequently, it is a helpful technique to define them separately in an external definition file (EDF), so that they can be referenced from any program or module using the metacommand $INCLUDE.

The subsection "EDF File Example" shows one such EDF file, Syslit.Edf, which defines several generally useful data types.

A program can then be written that uses procedures, functions, and literals from Syslit.Edf simply by including the line below.

```
(*  $INCLUDE:'Syslit.Edf'  *)
```

Once this is done, all the procedures, functions, constants, and literals can be used as if declared within the program itself.

### EDF FILE EXAMPLE

```
(*
   FILE:  Syslit.Edf
  SYStem LITeral External Definition File
  Generally useful Pascal types for CTOS interface
*)

TYPE

     ErcType    = WORD;
     FlagType   = BOOLEAN;
     FhType     = WORD;
     LfaType    = INTEGER4;
     ModeType   = WORD;
     POINTER    = ADS of WORD;
     QUAD       = INTEGER4;
```

```
CONST

    ercOk        = Ø;
    modeAppend   = RETYPE(WORD, 'ma');
    modeModify   = RETYPE(WORD, 'mm');
    modeRead     = RETYPE(WORD, 'mr');
    modeWrite    = RETYPE(WORD, 'mw');
```

Pascal data types are not totally adequate for use with the CTOS operating system; therefore, data types that are roughly equivalent were chosen for Syslit.Edf in the example above. The semantics of the data types used in Syslit.Edf are shown below:

| | |
|---|---|
| ErcType | 2-byte unsigned integer; contains error status returned from a CTOS facility. Error status of Ø is no error. |
| FlagType | 1-byte unsigned integer. Ø means flag is off, and 1 means flag is on. |
| FhType | 2-byte unsigned integer; contains a file handle (number) that uniquely identifies open files for the file system. |
| LfaType | 4-byte unsigned integer; contains a logical file address (number) that identifies an offset from the beginning of a file. |
| ModeType | 2-byte string; contains two characters that indicate a file's access mode for the file system. |
| POINTER | 4-byte segmented address; contains two words, of which the low word is the relative address within a segment, and the high word is the segment base address. |
| QUAD | 4-byte unsigned integer; contains a number in the range Ø to 4,294,967,295 (used for arithmetic involving logical file addresses). Note that INTEGER4 does not satisfy this range. |

Table F-1 shows the CTOS type and the equivalent Pascal type.

**Table F-1. Pascal Data Types for Use with CTOS.**

| CTOS Type | Equivalent Pascal Type |
|---|---|
| ercType | WORD |
| pbType or pointer | ADS of WORD |
| flagType | BOOLEAN {ØØh = false, Ø1h = true} |
| fhType | WORD |
| modeType | WORD |
| lfaType or quadType | ADS of WORD, or INTEGER4<br><br>ADS of WORD works here if the type is used for displacement only or if the math performed on the type is WORD math.<br><br>INTEGER4 is not strictly an Lfa or a Quad, since the most significant bit is used as a sign, but it works for positive numbers. |

## CTOS STRUCTURES AND PASCAL

Pascal word-aligns all fields in a record.

CTOS structures, however, are often not word-aligned. In these cases you can use the explicit offset syntax for record fields.

For example:

```
ExpDateTimeType = record
                  year[Ø0]:word;
                  month[Ø2]:byte;
                  monthDay[Ø3]:byte;
                  weekDay[Ø4]:byte;
                  hour[Ø5]:byte;
                  minute[Ø6]:byte;
                  second[Ø7]:byte;
                  end;
```

(See the subsection "Explicit Field Offsets" in Section 6, "Arrays, Records, and Sets," for a discussion.

## ACCESSING CTOS STRUCTURES FROM PASCAL: EXAMPLE

```
{This is an example of how to access CTOS struc-
tures from Pascal.  The program displays the OS
version found in the System Common Address Table
(SCAT), the amount of memory allocated to CTOS
found in the System Configuration Block (SCB), and
the total amount of memory in the workstation,
also found in the SCB.}

{$debug-}

Program TalkingToCTOS     (Output);


Type
  pbType     =  ads of word;
    {pointer to word}
  ppType     =  ads of pbType;
    {pointer to pointer}
  paraType   =  word;
    {a paragraph of memory is 16 words}

{Type definition for System Configuration Block,
below, describes memory as follows:


  saMemMax  ---------- top of memory
            |        |
  saMaxSL   ---------- top of short-lived memory
            |        |
            |        |
  saCurrSL  ---------- bottom of short-lived
            |        | memory
            |        |
  saCurrLL  ---------- top of long-lived memory
            |        |
  saMinLL   ---------- top of CTOS /
            |        | bottom of long-lived memory
            |        |
            ---------- bottom of memory (0:0)

Segment addresses point to paragraphs; they can be
multiplied by the paragraph size (16 bytes) to
determine the physical location in bytes of the
segment.

Note: When accessing CTOS structures whose fields
are not word aligned, explicit offsets must be
used in field definition, since Pascal will
otherwise word-align fields.}
```

```
SCBType = record
            SysBuildType     [00]:byte;
            OsType           [01]:byte;
            saMinLL          [02]:paraType;
            saCurrLL         [04]:paraType;
            saCurrSL         [06]:paraType;
            saMaxSL          [08]:paraType;
            saMemMax         [10]:paraType;
            end;
pSCBType  = ads of SCBType;

VersionType =  lstring(30);
  {version is an 'sb' string, a.k.a, lstring}

pVersionType = ads of VersionType;

{definitions of CTOS externals:}

Function GetpStructure (
    structCode          :word;
    ph                  :word;
    {partition handle}
    ppStructureRet      :ppType )     :word; extern;

Procedure CheckErc (
    erc                 :word ); extern;


Procedure DumpCTOSVersion     [public];
  Const oVersion = #254;
    {oVersion is the relative address of the
    pointer to the pointer to the version. It can
    be found in the System Common Address Table
    (SCAT) described in the CTOS Operating System
    Manual, Volume 2.  The segment address for all
    fields in the SCAT is zero}

    var
      pVersion :pVersionType;
        {pointer to CTOS version}
      Version  :VersionType;
    begin

    {GetpStructure takes as arguments a structure
    code or relative address of a structure
    defined in the SCAT, a partition handle (if
    zero then the handle of the partition the
    program is running in), and the address of the
    address to be returned}
```

```
    CheckErc (GetpStructure (oVersion, 0,
              ads pVersion));
    Version := pVersion^;
      {deference pointer to our lstring}

      {note: deferencing structure pointers
      requires the run time. Structure pointers
      can be deferenced without the run time on a
      field by field basis}

    Writeln ('CTOS version    ',Version);
    end;



Procedure DumpMemoryMap         [public];
  Const oSCB = #2C8;
  {relative address of the pointer to the System
  Configuration Block}

  var     sOsMemory,
          sMaxMemory,
          sParagraph    :integer4;
          pSCB          :pSCBType;
          {pointer to SCB}
  begin
  CheckErc (GetpStructure (oSCB, 0, ads pSCB));
  sParagraph := 16;
  sOsMemory  := sParagraph * pSCB^.saMinLL;
  sMaxMemory := sParagraph * pSCB^.saMemMax;
  Writeln ('OS memory     ', sOsMemory, ' bytes');
  Writeln ('Total memory ', sMaxMemory, ' bytes');
  end;



begin
    DumpCTOSVersion;
    DumpMemoryMap;
end.
```

## CONTROL OF THE VIDEO DISPLAY

You can control the video display using one of three different methods: Video Bytestreams, Direct Video Access (Video Access Method) through CTOS, or the Forms package.

Using Forms is described in detail in the Forms Manual. Examples of using Forms with Pascal are available as application notes from technical support.

Section 19, "Video," in the CTOS Operating System Manual describes the Video Access Method (VAM) in detail. In addition, an example showing the use of VAM is included at the end of this section. Direct Video Access has the advantage that it does not use the Pascal run-time library.

The remainder of this section describes video byte streams and shows how to control the video display from your Pascal program by writing a multibyte escape sequence to the display. This allows you to use WRITE and WRITELN to send an escape sequence to the screen in Pascal in the same way that you can use OpenByteStream in CTOS. In this way, a program can

- control character attributes (blinking, reverse video, underscoring, half-bright)
- control screen attributes (reverse video, half bright)
- fill a rectangle with a single character
- control scrolling of lines
- direct video display output to any frame
- control pausing between full frames of data
- control the keyboard LED indicators
- erase to the end of the current line or frame

A multibyte escape sequence consists of the video display escape character, a command character, and parameters. The video display escape character is CHR(255). To print an escape character, precede it with another escape character.

The following pages give the format for escape sequences that control the various features of the video display. Note that these formats show the asterisk (*) as the concatenation operator, but the asterisk can only be used to create constant string expressions. Variable string expressions with concatenation, should use the LSTRING intrinsic CONCAT.

## ERROR CONDITIONS IN ESCAPE SEQUENCES

An escape character sequence is in error if the command characters or parameters are unrecognized or the parameters are inconsistent.

The following program turns on the cursor, writes the message "This is a test," and waits for input:

```
PROGRAM Test (INPUT, OUTPUT);
VAR
  LS : LSTRING (128);

BEGIN
  LS := CHR(255) * 'vn';
  Write (LS, 'This is a test');
  ReadLn;
END.
```

## VIDEO DISPLAY COORDINATES

Pascal interprets some parameters as x and y coordinates on the video display.

A value of 255 for x or y specifies, respectively, the last column or line of the frame.

If the value of x or y is less than 255 and greater than the last column or line, then the parameters are in error.

## CONTROLLING CHARACTER ATTRIBUTES: THE 'A' COMMAND

Two formats are available for giving the 'A' command.

### Format 1

```
CHR(255) * 'A<parameter>'
```

where

<parameter>
is a character in the range A to P.

Format 1 is used to enable or disable character attributes for characters following the escape sequence. Table F-2 shows the attributes enabled or disabled for each escape sequence using the 'A' command.

A yes in Table F-2 indicates that the attribute is enabled; otherwise, it is disabled.

**Table F-2. Character Attributes.**

| Mode | Blink | Reverse | Underline | Half-bright |
|---|---|---|---|---|
| CHR(255) * 'AA' | no | no | no | no |
| CHR(255) * 'AB' | no | no | no | yes |
| CHR(255) * 'AC' | no | no | yes | no |
| CHR(255) * 'AD' | no | no | yes | yes |
| CHR(255) * 'AE' | no | yes | no | no |
| CHR(255) * 'AF' | no | yes | no | yes |
| CHR(255) * 'AG' | no | yes | yes | no |
| CHR(255) * 'AH' | no | yes | yes | yes |
| CHR(255) * 'AI' | yes | no | no | no |
| CHR(255) * 'AJ' | yes | no | no | yes |
| CHR(255) * 'AK' | yes | no | yes | no |
| CHR(255) * 'AL' | yes | no | yes | yes |
| CHR(255) * 'AM' | yes | yes | no | no |
| CHR(255) * 'AN' | yes | yes | no | yes |
| CHR(255) * 'AO' | yes | yes | yes | no |
| CHR(255) * 'AP' | yes | yes | yes | yes |

**Format 2**

```
CHR(255) * 'AZ'
```

Format 2 is used to enable a mode whereby writing a character into a character position does not change the character attributes of that character position.

**CONTROLLING SCREEN ATTRIBUTES: THE 'H' AND 'R' COMMANDS**

**Format 1**

```
CHR(255) * 'H<parameter>'
```

where

<parameter>
is N or F.

Format 1 is used to turn the half bright attribute on if the <parameter> is N or off if the <parameter> is F.

**Format 2**

```
CHR(255) * 'R<parameter>'
```

where

<parameter>
is N or F.

Format 2 is used to turn the reverse video attribute on if the <parameter> is N or off if the <parameter> is F.

**CONTROLLING CURSOR POSITION AND VISIBILITY: THE 'C' AND 'V' COMMANDS**

**Format 1**

```
CHR(255) * 'C' * CHR(<Xposition>)
* CHR(<Yposition>)
```

where

<Xposition> and <Yposition>
are integer expressions.

Format 1 is used to position the cursor at coordinates (<Xposition>,<Yposition>).

**Format 2**

```
CHR(255) * 'V'<parameter>
```

where

<parameter>
is N or F.

Format 2 is used to make the cursor visible if the <parameter> is N or to make the cursor invisible if the <parameter> is F.

## FILLING A RECTANGLE: THE 'F' COMMAND

```
CHR(255) * 'F' * <character>
* CHR(<Xposition>)
* CHR(<Yposition>)
* CHR(<width>) * CHR(<height>)
```

where

<character>
is any character;

<Xposition>, <Yposition>, <width>, and <length>
are integer expressions.

The 'F' command is used to fill a rectangle on the video display with <character>. The currently enabled character attributes are given to each character in the rectangle. A <character> always specifies a character in the standard character set.

The coordinates (<Xposition>,<Yposition>) specify the upper left corner of the rectangle. A value of 255 for <width> and <height> specifies, respectively, the remaining width or height of the frame.

## CONTROLLING LINE SCROLLING: THE 'S' COMMAND

```
CHR(255) * 'S'
* CHR(<firstline>)
* CHR(<lastline>)
* CHR(<count>) * '<direction>'
```

where

<direction>
is D or U.

If the <direction> is D, the 'S' command is used to scroll down a portion of the frame beginning at line <firstline> and extending to (but not including) <lastline>. The <count> lines are scrolled and the top <count> lines of the frame portion are filled with blanks.

If the <direction> is U, the 'S' command is used to scroll up a portion of the frame beginning at line <lastline> and extending to (but not including) <firstline>. The <count> lines are scrolled and the bottom <count> lines of the frame portion are filled with blanks.

**DIRECTING VIDEO DISPLAY OUTPUT: THE 'X' COMMAND**

```
CHR(255) * 'X' * CHR(<frame>)
```

The 'X' command is used to direct video output to the <frame>'th frame of the video display.

The video display is divided into frames. (See Section 19, "Video," in the CTOS Operating System Manual for a discussion of video frames.)

The main frame is the default frame.

If <frame> is 1, the 'X' command is used to direct video output to the Status Frame at the top of the video display.

If <frame> is 2, the output is directed to the line that separates the Status Frame from the main frame.

**CONTROLLING PAUSING BETWEEN FULL FRAMES: THE 'P' COMMAND**

```
CHR(255) * 'P<parameter>'
```

where

<parameter>
is N or F.

If the <parameter> is N, the 'P' command is used to enable the pause facility. When the pause facility is enabled and further output to the frame would cause data to be scrolled off the top of the frame, the message:

Press NEXT PAGE or SCROLL UP to continue

is displayed on the last line of the frame.

If the <parameter> is F, the 'P' command is used to disable the pause facility.

**CONTROLLING THE KEYBOARD LED INDICATORS: THE 'I' COMMAND**

CHR(255) * 'I<parameter>'

where

<parameter>
is 1, 2, 3, 8, 9, Ø, or T.

The 'I' command is used to turn on an LED indicator on the keyboard according to the Table F-3.

**Table F-3. LED Parameters.**

| <parameter> | Key |
|---|---|
| 1 | F1 |
| 2 | F2 |
| 3 | F3 |
| 8 | F8 |
| 9 | F9 |
| Ø | F1Ø |
| T | OVERTYPE |

**ERASING TO THE END OF THE LINE OR FRAME: THE 'E' COMMAND**

CHR(255) * 'E<parameter>'

where

<parameter>
is L or F

If the <parameter> is L, the 'E' command is used to erase to the end of the line.

If the <parameter> is F, the 'E' command is used to erase to the end of the frame.

Erasing sets characters to spaces and turns off all character attributes.

**EXAMPLE OF PASCAL CONTROL OF THE VIDEO DISPLAY**

```
{$debug-}

Program VideoSample_PascalRuntime (Input, Output);


    {Video sample using multibyte esacpe sequences
    and the Pascal run time. The program clears
    frame zero, paints a screen, accepts input
    from three field, and displays/scrolls the
    input on the lower portion of the screen.}

Const
    cMaxFields    =3;
    bEsc          =chr(#ff); {escape character}
    bReverse      =chr(#45);
    bHalfBright   =chr(#42);
    bNormal       =chr(#41);
    bSpace        =chr(#20);

Type
    fieldType       =lstring(10);
    fieldDescType   =record
                     colLabel        :byte;
                     rowLabel        :byte;
                     colInput        :byte;
                     rowInput        :byte;
                     lsLabel         :fieldType;
                     lsInput         :fieldType;
                     cbInput         :word;
                     end;

Var   [public]
    iField        :word;
    lsSpace       :fieldType;
    rgFieldDesc   :array [wrd(1)..wrd(cMaxFields)]
                  of fieldDescType;
```

```
Value
    lsSpace                         := '          ';
    rgFieldDesc[1].colLabel         := 30;
    rgFieldDesc[1].rowLabel         :=  4;
    rgFieldDesc[1].colInput         := 40;
    rgFieldDesc[1].rowInput         :=  4;
    rgFieldDesc[1].lsLabel          := 'Field 1';
    rgFieldDesc[1].cbInput          := 10;
    rgFieldDesc[2].colLabel         := 30;
    rgFieldDesc[2].rowLabel         :=  5;
    rgFieldDesc[2].colInput         := 40;
    rgFieldDesc[2].rowInput         :=  5;
    rgFieldDesc[2].lsLabel          := 'Field 2';
    rgFieldDesc[2].cbInput          := 10;
    rgFieldDesc[3].colLabel         := 30;
    rgFieldDesc[3].rowLabel         :=  6;
    rgFieldDesc[3].colInput         := 40;
    rgFieldDesc[3].rowInput         :=  6;
    rgFieldDesc[3].lsLabel          := 'Field 3';
    rgFieldDesc[3].cbInput          := 10;


Procedure PutAttrs (
    bAttr       :char)                          [public];
    var
        escSeq :string(3);
    begin
    escSeq[1] := bEsc;
    escSeq[2] := 'A';
    escSeq[3] := bAttr;
    Write (escSeq);
    end;

Procedure PutField (
    iCol,
    iRow        :byte;
    LsField     :fieldType)     [Public];
    var
        escSeq :string(4);
    begin
    escSeq[1] := bEsc;
    escSeq[2] := 'C';
    escSeq[3] := chr(iCol);
    escSeq[4] := chr(iRow);
    PutAttrs (bHalfBright);
    Write (escSeq);
    Write (lsField);
    PutAttrs (bNormal);
    end; {PutField}
```

```
Procedure InitVideo              [Public];
    begin
    {position cursor at top of frame and clear
    frame}
    PutField (#00,#00,' ');
    Write (bEsc * 'EF');
    input.trap := true;
    end;    {InitVideo}


Function GetField (
    iCol,
    iRow         :byte;
    Var lsField :fieldType;
    cbText       :word)   :boolean        [Public];
    var
        escSeq   :string(4);
    begin
    escSeq[1] := bEsc;
    escSeq[2] := 'C';
    escSeq[3] := chr(iCol);
    escSeq[4] := chr(iRow);
    Write (escSeq);
    PutAttrs (bNormal);
    Write(lsSpace);
    Write (escSeq);
    Readln(lsField);
    if input.errs <> 0 then
         {e.g. finish is depresssed}
        getField := false
            else getField := true;
    end; {GetField}

Procedure Scroll         [public];
    var
        escSeq    :string(6);
    begin
    {Scroll rows 12 to 27 up by 1 line}
    escSeq[1] := bEsc;
    escSeq[2] := 'S';
    escSeq[3] := chr(12);
    escSeq[4] := chr(28);
    escSeq[5] := chr(01);
    escSeq[6] := 'U';
    Write(escSeq);
    end;
```

```
begin
    InitVideo;
    for iField := 1 to cMaxFields do
        with rgFieldDesc[iField] do
            PutField (colLabel,rowLabel,
                      lsLabel);
    iField := 1;
    While true do
        begin
        with rgFieldDesc[iField] do
            begin
            lsInput := lsSpace;
            if not GetField (colInput,rowInput,
                             lsInput,cbInput) then
                break;
            Scroll;
            PutField (40,27, lsInput);
            if iField < cMaxFields then
                iField := iField + 1
                    else iField := 1;
            end;
        end;
end.
```

**EXAMPLE OF CTOS CONTROL OF THE VIDEO DISPLAY USING VAM**

```
{$debug-}

Program VideoSample_Vam;


    {Video sample using VAM and VDM. This program
    does not make Pascal run-time calls and may be
    linked with Pasmin.Obj.  The program resets
    the video, paints an initial screen, accepts
    input from three fields, and displays and
    scrolls the input on the lower portion of the
    screen.

    An alternative method of dealing with the
    video and keyboard is to use the Forms
    package. The advantage of using Forms is that
    the application need not deal with physical
    screen coordinates, and input/output calls
    perform type conversion.}


Const
    cMaxFields        =3;
    ercOk             =Ø;
    bAttrInput        =#Ø4;    {reverse video}
    bAttrOutput       =#Ø1;    {half-bright}
    bCr               =#Øa;    {RETURN/NEXT key}
    bGo               =#1b;    {GO key}
    bBackSpace        =#Ø8;    {BACKSPACE key}
    bFinish           =#Ø4;    {FINISH key}
    bSpace            =#2Ø;    {SPACE key}

Type
    pbType          =ads of word;
    pbyType         =ads of byte;
    prgbType        =ads of array
                     [wrd(1)..wrd(2)] of byte;
    fieldDescType =record
                     colLabel      :word;
                     rowLabel      :word;
                     colInput      :word;
                     rowInput      :word;
                     lsLabel       :lstring(2Ø);
                     rgbInput      :array
                      [wrd(1)..wrd(2Ø)] of byte;
                     cbInput       :word;
                         end;
```

```
Var     [public]
    erc,
    iField          :word;
    lsSpace         :lstring(20);
    rgFieldDesc     :array
                    [wrd(1)..wrd(cMaxFields)]
                    of fieldDescType;

Value
    lsSpace                 := '                         ';
    rgFieldDesc[1].colLabel     := 30;
    rgFieldDesc[1].rowLabel     :=  4;
    rgFieldDesc[1].colInput     := 40;
    rgFieldDesc[1].rowInput     :=  4;
    rgFieldDesc[1].lsLabel      := 'Field 1';
    rgFieldDesc[1].cbInput      := 10;
    rgFieldDesc[2].colLabel     := 30;
    rgFieldDesc[2].rowLabel     :=  5;
    rgFieldDesc[2].colInput     := 40;
    rgFieldDesc[2].rowInput     :=  5;
    rgFieldDesc[2].lsLabel      := 'Field 2';
    rgFieldDesc[2].cbInput      := 10;
    rgFieldDesc[3].colLabel     := 30;
    rgFieldDesc[3].rowLabel     :=  6;
    rgFieldDesc[3].colInput     := 40;
    rgFieldDesc[3].rowInput     :=  6;
    rgFieldDesc[3].lsLabel      := 'Field 3';
    rgFieldDesc[3].cbInput      := 10;



{VAM, VDM, Keyboard Management external
definitions:}

Function ResetVideo (
    nCols,
    nLines      :word;
    fAttr       :boolean;
    bSpace      :byte;
    psMapRet    :pbtype)    :word;      extern;
Function InitVidFrame (
    iFrame,
    iCol,
    iRow,
    nWidth,
    nHeight     :word;
    borderDesc,
    borderChar,
    borderAttr  :byte;
    fDblh,
    fDblw       :boolean)   :word;      extern;
```

```
Function InitCharMap (
    pMap        :pbtype;
    sMap        :word)          :word;      extern;
Function SetScreenVidAttr (
    iAttr       :word;
    fAttr       :boolean)       :word;      extern;
Function ResetFrame (
    iFrame      :word)           :word;     extern;
Function PutFrameChars (
    iFrame,
    iCol,
    iRow        :word;
    pbText      :pbType;
    cbText      :word)          :word;      extern;
Function PutFrameAttrs (
    iFrame,
    iCol,
    iRow        :word;
    bAttr       :byte;
    nPos        :word)          :word;      extern;
Function ScrollFrame (
    iFrame,
    iLineStart,
    iLineMax,
    cLines      :word;
    fUp         :boolean)       :word;      extern;
Function PosFrameCursor (
    iFrame,
    iCol,
    iLine       :word)          :word;      extern;

Function ReadKbd (
    pByte       :pbyType)       :word;      extern;
Function Beep                   :word;      extern;

Procedure CheckErc (
    erc         :word)          ;extern;
Procedure Exit                  ;extern;



Function InitVideo (
    nCol,
    nRow        :word)  :word               [Public];
    var
        pCharMap    :pbType;
        sCharMap    :word;
```

```
    begin
    InitVideo := ercOk;
    erc := ResetVideo (nCol,nRow,true,#20,
        ads sCharMap);
    if erc<>ercOk then
        begin InitVideo:=erc; return; end;
    erc := InitVidFrame
(0,0,0,nCol,nRow,#00,#20,#00,false,false);
    if erc<>ercOk then
        begin InitVideo:=erc; return; end;
    pCharMap.r:=0;
    pCharMap.s:=0;
    {null pCharMap means use existing character
    map}
    erc := InitCharMap (pCharMap, sCharMap);
    if erc<>ercOk then
        begin InitVideo:=erc; return; end;
    erc := SetScreenVidAttr (1,true);
    if erc<>ercOk then
        begin InitVideo:=erc; return; end;
    erc := resetFrame (0);
    if erc<>ercOk then InitVideo:=erc;
    end;    {InitVideo}


Function PutField (
    iCol,
    iRow    :word;
    pbText  :pbType;
    cbText  :word)          :word       [Public];
    begin
    PutField := ercOk;
    erc := PutFrameAttrs
           (0, iCol, iRow, bAttrOutput, cbText);
    if erc<>ercOk then
        begin PutField := erc; return; end;
    erc := PutFrameChars
              (0, iCol, iRow, pbText, cbText);
    if erc<>ercOk then
        PutField := erc;
    end; {PutField}


Function GetField (
    iCol,
    iRow    :word;
    pbText  :prgbType;
    cbText  :word)          :word       [Public];
    Var
        iPos,
        ib   :word;
        b    :byte;
```

```
begin
GetField := ercOk;
erc := PutFrameAttrs
       (Ø, iCol, iRow, bAttrInput, cbText);
if erc<>ercOk then
    begin GetField := erc; return; end;
erc := PutFrameChars
      (Ø, iCol, iRow, ads lsSpace[1], cbText);
if erc<>ercOk then
    begin GetField := erc; return; end;
iPos := 1;
While true do
    begin
    erc := PosFrameCursor
          (Ø, (iCol + iPos - 1), iRow);
    if erc<>ercOk then
        begin GetField := erc; return; end;
    erc := ReadKbd (ads b);
    case b of
        bCr, bGo    :break;
        bFinish     :exit;
        bBackSpace :if iPos > 1 then
                      begin
                      pbText^[iPos] := bSpace;
                      iPos := iPos - 1;
                      erc := PutFrameChars
                         (Ø, (iCol + iPos - 1),
                         iRow, ads #2Ø,1) ;
                      if erc<>ercOk then
                      begin GetField:=erc;
                         return; end;
                         end
                              else erc:=beep;
        otherwise      if iPos > cbText then
                         erc := beep
                         else
                         begin
                         erc := PutFrameChars
                         (Ø, (iCol + iPos - 1),
                          iRow, ads b,1) ;
                         if erc<>ercOk then
                           begin GetField:=erc;
                            return; end;
                         pbText^[iPos] := b;
                         iPos := iPos + 1;
                    end;
        end; {case}
    end; {while}
```

```
    for ib:= iPos to cbText do
        pbText^[ib] := bSpace;
        {pad with blanks}

    erc := PutFrameAttrs
             (Ø, iCol, iRow, bAttrOutput, cbText);
    if erc<>ercOk then
        GetField := erc;
    end; {GetField}


begin
    CheckErc (InitVideo (8Ø,28));
    {initialize video for 8Ø columns, 28 lines}

    for iField := 1 to cMaxFields do
        with rgFieldDesc[iField] do
            CheckErc (PutField (colLabel,rowLabel,
                      ads lsLabel[1],
                       wrd(lsLabel.len)));

    iField := 1;
    While true do
        begin
        with rgFieldDesc[iField] do
         begin
         CheckErc (GetField (colInput,rowInput,
                    ads rgbInput,cbInput));
         CheckErc (ScrollFrame (Ø,12,28,1,true));
         CheckErc (PutField (4Ø,27, ads
                   rgbInput,cbInput));
            if iField < cMaxFields then
                iField := iField + 1
                    else iField := 1;
            end;
        end;
end.
```

# APPENDIX G: INTERNAL REPRESENTATIONS OF DATA TYPES

## INTEGER AND WORD

INTEGER values are 16-bit twos complement numbers, but a subrange requiring 8 bits or less (in the range -127..127) is allocated an 8-bit byte. WORD values are 16-bit unsigned numbers, but a WORD subrange in the range Ø..255 is allocated an 8-bit byte. For 16-bit INTEGER and WORD values, the least significant byte has the lower, even address.

## INTEGER4

INTEGER4 values are 32-bit twos complement numbers, with the least significant byte at the lowest, even address and more significant bytes at increasing addresses. There are no subranges for INTEGER4 (as there are for INTEGER2.)

IEEE 4-byte real numbers have a sign bit, 8-bit excess 127 binary exponent, and a 24-bit mantissa. The mantissa represents a number between 1.Ø and 2.Ø. Since the high-order bit of the mantissa is always 1, it is not stored in the number. This representation gives an exponent range of 1Ø**38 and 7 digits of precision. The maximum real number is normally 1.7Ø1411E38.

For both INTEGER4 and REAL numbers, a number with an exponent of all zeros is considered zero. An exponent of all ones is a flag for an invalid real number, or "not a number" (NaN).

## REAL

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

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

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

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

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

As an extension to standard Pascal, the exponent character can be "D" or "d" as well as "E" or "e", for example, 12.34d56. Note that, the D or d exponent character does not indicate double precision, as it does in FORTRAN.

For both INTEGER4 and REAL numbers, a number with an exponent of all zeros is considered zero. An exponent of all ones is a flag for an invalid real number or NaN.

## CHAR, BOOLEAN, AND ENUMERATED TYPES

CHAR values and BOOLEAN values take 8 bits. CHAR values correspond to the ASCII collating sequence. For BOOLEAN values, FALSE is Ø and TRUE is 1. The low-order bit (bit Ø) is generally used to check this value. Bits 1 through 7 are presumed to be Ø.

Enumerated values take 8 bits if 256 or fewer values are declared; otherwise 16 bits are declared. Values are assigned starting at Ø. Subrange values take either 8 or 16 bits.

## REFERENCE TYPES

Pointer values currently take 16 bits. A pointer is a default data segment offset. A pointer to a super array type is followed by the bounds (see the subsection "Super Arrays" below), increasing the length of the pointer value (DS/SS).

ADR and ADS are offset addresses and segmented addresses, respectively. For segmented addresses, the offset is the lower address, and the segment follows.

The heap contains heap blocks, which may be allocated or free. A heap block contains a header WORD, with a 15-bit length (in WORDs) and the lower-order bit, which is ON for free blocks and OFF for allocated blocks. The starting and ending heap addresses are WORD variables in BEGHQQ and ENDHQQ.

## PROCEDURAL AND FUNCTIONAL PARAMETERS

Procedural parameters contain a reference to the location of the procedure or function along with a reference to the upper frame pointer.

The parameter always contains two words, in one of two formats:

- In the first format, the first word contains the actual routine's address (a local code segment offset), and the second word contains the upper frame pointer. The upper frame pointer is zero if the actual routine is not nested in a procedure or function and, therefore, the routine has no upper frame pointer.

- In the second format, used for segmented address targets, the first word is zero and the second word contains a data segment offset address. This is an offset to two words in the constant area that contain the segmented address of the actual routine. There is never an upper frame pointer in this case.

## SUPER ARRAYS

Representation of a super array type is similar whether it is a reference parameter or the referent of a pointer. First comes the address (reference parameter) or pointer value, which is either 2 or 4 bytes long. Following the address are the upper bounds, which are signed or unsigned 16-bit quantities. The bounds occur in the same order as they are declared. A pointer value to a super array type is normally longer than other pointers, since the upper bounds are included.

## SETS

The number of bytes allocated for a SET is:

```
(ORD (upperbound) DIV 16) * 2 + 2
```

This is always an even number from 2 to 32 bytes. For example, SET of 'A'..'Z' requires 12 bytes. Internally, a set consists of an array of bits, with one bit for every possible ORD value from Ø to the upper bound. Bits in a byte are accessed starting with the most significant bit. The occurrence of a given ORD value as an element of a set implies the bit is 1, and the byte and bit position of a given ORD value of any set is the same. For example, the ORD value of 'A' is 65, and the second bit (that is, 2#Ø1ØØØØØØ) of the ninth byte in a set is 1 if 'A' is in the set.

## FILES

A FILE type in a program is a record called a file control block (of type FCBFQQ) in the file unit. The initial portion of the FCBFQQ record is standard for all files, but the remainder is available for use by the file system. The end of the FCB contains the current buffer variable. The internal form of a file varies depending on the target file system.

## STRUCTURES

For arrays and records, the internal form is comprised of the internal forms of the components, in the same order as in the declaration. Arrays, records, variants, sets, and files always start on a word boundary. In any case, variables cannot be allocated more than MAXWORD (64K) bytes.

A PACKED type has the same representation as an unpacked one.

A variable or component 16 bits or larger is always aligned on a word boundary; therefore, it always has an even byte address. The only exception is when explicit field offsets are given by the user in a program.

An 8-bit variable is also aligned on a word boundary, but an 8-bit component of a structure (array or record) is aligned on a byte boundary, which can be at an even or odd address. Currently an array of 8-bit variables starts on a word boundary.

## EXAMPLES SHOWING THE USE OF MODULES AND UNITS

The following two examples both perform the same job, converting a temperature in Celsius to Fahrenheit. Both use an external function.

Example 1 uses a module to declare the function, while Example 2 uses a unit.

All the examples in this Appendix were compiled and run with 9.Ø level software.

### EXAMPLE 1

The two files are separately compiled, then linked to create the run file Pel.Run. Pel.Pas contains the main program, and Pe2.Pas contains a module that declares a function that changes temperature from Celsius to Fahrenheit.

Instructions for compiling, linking, and running the two compilands are given in the subsection "Instructions for Compiling, Linking, and Running Example 1."

**Main Program: Pel.Pas**

```
(* Files Pel.Pas and Pe2.Pas must be compiled
separately and linked together.*)

(* This program converts Celsius temperature to
Fahrenheit.  It prompts the user to enter the
Celsius temperature, then converts that to
Fahrenheit, and displays the result on the screen.
It then prompts the user for another Celsius
temperature, and so on. The program terminates
when the user enters a number less than -2ØØ.
*)

(*The program uses an external function,
Fahrenheit, to compute the Fahrenheit temperature.
That function is declared in a separate compiland
in the file Pe2.Pas
*)
```

```
Program  CelsiusToFahrenheit(Input,Output);

VAR celsTemp : REAL;

FUNCTION Fahrenheit (celsius : REAL) : REAL;
EXTERN;

BEGIN

REPEAT

(* Prompt the user for input.*)

write ('Enter Celsius temperature');
write ('       ( -200 or less to exit): ');

(* Read the response.*)

readln(celsTemp);

IF celsTemp <= -200 THEN BREAK; (* Check for
                                   sentinel value*)

(* Convert Celsius temperature to Fahrenheit and
display the result.*)

writeln;
writeln(celsTemp:6:3,' C = ',
                Fahrenheit(celsTemp):6:3,' F');
writeln;

UNTIL FALSE

END.
```

**Module: Pe2.Pas**

```
(* Files Pel.Pas and Pe2.Pas must be compiled
separately and linked together.  *)

(*This file contains a module declaring the
function Fahrenheit.
*)

module Fah;

FUNCTION Fahrenheit(cels:REAL) : REAL;

(* This function converts Celsius temperature to
Fahrenheit.
ON ENTRY: cels is temperature in degrees Celsius.
RETURN: The function returns temperature in
degrees Fahrenheit.
*)

BEGIN

Fahrenheit := cels * (9/5) + 32

END; (* End of Fahrenheit.*)

END. (* End of module.*)
```

**Instructions for Compiling, Linking, and Running Example 1**

To invoke the compiler, type "Pascal" into the Executive command form. Complete the Pascal command form as shown below, then press GO:

```
Pascal
  Source file          Pel.Pas
  [Object file]
  [List file]
  [Object list file]
```

After the program has compiled, compile the module the same way, but complete the command form as shown below:

```
Pascal
  Source file          Pe2.Pas
  [Object file]
  [List file]
  [Object list file]
```

Then link the resulting two object files, Pel.Obj and Pe2.Obj. The Linker is invoked through the Executive, by typing "Link" (or as many letters as required to make the command unique) into the Executive command form. Complete the form as shown below:

```
Link
  Object modules              Pel.Obj Pe2.Obj
  Run file                    Pel.Run
  [List file]
  [Publics?]
  [Line numbers?]
  [Stack size]
  [Max memory array size]
  [Min memory array size]
  [System build?]
  [Version]
  [Libraries]
  [DS Allocation]
  [Symbol file]
```

The resulting run file, Pel.Run, can be invoked by completing the Run command form as shown below. Remember, to terminate the program, enter a Celcius temperature of less than -200.

```
Run
  Run file          Pel.Run
  [Case]
  [Parameter 1]
  [Parameter 2]
  [Parameter 3]
  [Parameter 16]
```

**EXAMPLE 2**

The three files shown below perform the same job as the files shown in Example 1.

Pe3.Pas contains the main program, and Pe4.Pas contains a unit that declares a function that changes temperature from Celsius to Fahrenheit.

These two files are separately compiled, then linked to create the run file Pe3.Run.

The interface file, Pei.Inf (the third file shown below), is used by both Pe3.Pas and Pe4.Pas. It is not compiled separately.

Instructions for compiling, linking, and running the two compilands are given in the subsection "Instructions for Compiling, Linking, and Running Example 2."

**Main Program: Pe3.Pas**

```
(* This file, Pe3.Pas, must be compiled separately
and linked together with Pe4.Pas.  Both Pe3.Pas
and Pe4.Pas use an interface in the file Pei3.Inf.
The File Pei3.Inf cannot be compiled separately.*)

(* Pe3.Pas and Pe4.Pas implement the same program
as the files Pel.Pas and Pe2.Pas in Example 1, but
here we use a unit instead of a module to
implement the function Fahrenheit.*)

(* This program converts Celsius temperature to
Fahrenheit.  It prompts the user to enter the
Celsius temperature, then it converts it to
Fahrenheit, and displays the result on the screen.
It then prompts the user for another Celsius
temperature, and so on. The program terminates
when the user enters a number less then -2ØØ.
*)

(*The program uses an external function,
Fahrenheit, to compute the Fahrenheit temperature.
That function is declared in a separate compiland
in the file Pe4.Pas
*)
```

```
(* $INCLUDE:'Pei3.Inf'     --- interface file.*)

Program  CelsiusToFahrenheit(Input,Output);

USES Fah(Fahrenheit);

VAR celsTemp : REAL;

BEGIN

REPEAT

(* Prompt the user for input.*)

write ('Enter Celsius temperature');
write ('       ( -200 or less to exit): ');

(* Read the response.*)

readln(celsTemp);

IF celsTemp <= -200 THEN BREAK; (* Check for
                                   sentinel value*)

(* Convert Celsius temperature to Fahrenheit and
display the result.*)

writeln;
writeln(celsTemp:6:3,' C = ',
                Fahrenheit(celsTemp):6:3,' F');
writeln;

UNTIL FALSE

END.
```

**Unit: Pe4.Pas**

```
(* Pe4.Pas must be compiled separately and linked
together with Pe3.Pas.  Both files use an
interface in the file Pei3.Inf.  File Pei3.Inf
cannot be compiled separately.*)

(*This file contains an implementation of unit Fah
*)

(* $INCLUDE:'Pei3.Inf'    --- interface file.*)

IMPLEMENTATION OF Fah;

FUNCTION CompFah;   (* (cels : REAL) : REAL *)

(* This function converts Celsius temperature to
Fahrenheit.
ON ENTRY: cels is temperature in degrees Celsius.
RETURN:  The function returns temperature in
degrees Fahrenheit.
*)

BEGIN

CompFah := cels * (9/5) + 32

END; (* End of CompFah.*)

END. (* End of module.*)
```

**Interface: Pei3.INF**

```
(* Interface for the unit Fah.  This file is
INCLUDEd into the files Pe3.Pas and Pe4.Pas.  This
file is not a compiland (it is not compiled
separately).
*)

INTERFACE (2);   (* 2 is a version number.*)

UNIT Fah(CompFah);

FUNCTION CompFah(cels : REAL) : REAL;

END;
```

**Instructions for Compiling, Linking, and Running Example 2**

Invoke the compiler, as described in the subsection "Instructions for Compiling, Linking, and Running Example 1," above, and compile Pe3.Pas and Pe4.Pas each separately. Complete the command form as shown below:

```
Pascal
  Source file          Pe3.Pas
  [Object file]
  [List file]
  [Object list file]
```

```
Pascal
  Source file          Pe4.Pas
  [Object file]
  [List file]
  [Object list file]
```

Then link the resulting two object files, Pe3.Obj and Pe4.Obj, by completing the Linker command form as shown below:

```
Link
  Object modules              Pe3.Obj Pe4.Obj
  Run file                    Pe3.Run
  [List file]
  [Publics?]
  [Line numbers?]
  [Stack size]
  [Max memory array size]
  [Min memory array size]
  [System build?]
  [Version]
  [Libraries]
  [DS Allocation]
  [Symbol file]
```

The resulting run file, Pe3.Run, can be invoked by completing the Run command form as shown below. Remember, to terminate the program, enter a Celsius temperature of less than -200.

```
Run
  Run file          Pe3.Run
  [Case]
  [Parameter 1]
  [Parameter 2]
  [Parameter 3]
  [Parameter 16]
```

## EXAMPLE 3: BINARY TREE SEARCH

The following example shows a more complicated Pascal program than the examples given above. The program reads a file of characters, orders them (by their ASCII value), and prints them out in order. It stores the characters in an ordered binary tree and traverses the tree in order. Characters are read until it reaches the end of file or a period character (.) whichever comes first. The program uses an additional program parameter, Keyfile, as well as the file Input and Output.

The entire example consists of two compilands (a main program and a module that defines procedures) and an $INCLUDEd file that is not compiled separately.

Instructions for compiling, linking, and running the program appear below.

**MAIN PROGRAM: MAINTREE.PAS**

```
(* This file has the main program for the trees
example. The program reads a file of keys, builds
an ordered binary tree out of them, then traverses
the tree in order, displaying the keys.
*)

PROGRAM DisplayOrderedKeys(input,output,keyFile);

(* $INCLUDE:'Tree.Dcl' --- TYPE declarations.*)

VAR keyFile: KeyFileType; (* input file of keys.*)

(* External procedures and functions.*)

FUNCTION BuildTree (VAR keyFile : KeyFileType) :
                              TreeNodePtr;EXTERN;

(*This function builds a tree and returns
 a pointer to the tree.
PARAMETER: keyFile --- the file where the keys
           are.
RETURN: the function returns a pointer to the tree
        built.
*)


PROCEDURE TraverseTreeInOrder(root: TreeNodePtr;
          PROCEDURE Action(key:KeyType)); EXTERN;

(* This procedure traverses a tree in order while
calling a procedure to process each key.
PARAMETERS: root --- pointer to tree root,
            Action --- procedure to process each
            key.
*)

PROCEDURE DisplayKey(key:KeyType); EXTERN;

(* This procedure displays a key on the screen.
PARAMETER: key --- key to display.
*)

(* Internal procedure.*)
```

```
PROCEDURE DisplayTree(root:TreeNodePtr);

(* This procedure displays the keys ordered by
their value on the screen. It writes a heading,
then the keys.
PARAMETER : root --- pointer to the tree root.
*)

BEGIN

(* Write the heading.*)

writeln;
writeln('   ORDERED KEYS');
writeln;

(*Display the keys.*)

TraverseTreeInOrder(root,DisplayKey);

writeln                (* New line at the end*)

END;

BEGIN (* Main program*)

reset(keyFile);

DisplayTree(BuildTree(keyFile));

writeln;
writeln('Program terminated.')

END.
```

**MODULE: TREEMODULE.PAS**

```
(* This module contains procedures to build and
display trees.*)

module trees[];

(* $INCLUDE:'Tree.Dcl' *)

CONST
    SentinKey = '.'; (* Sentinel key value.*)

FUNCTION GetNewNode(VAR root:TreeNodePtr;
                    key: KeyType) : TreeNodePtr;

(* This function finds a place in a tree where a
key should be inserted, creates a node for
the key and inserts the node into the tree.
It does not fill the node fields.
PARAMETERS: root --- a pointer to the tree root,
            key --- the key.
RETURN: The function returns a pointer to the new
        node.
Note that root can be changed if the tree is
empty. *)

BEGIN

IF root = NIL          (* If tree is empty*)
THEN
    BEGIN
    new(root);         (* root points to new node*)
    GetNewNode := root     (* return the pointer*)
    END

ELSE                        (* tree is not empty *)

    IF key <= root^.nodeKey
    THEN                  (* Insert new node into*)
                          (*     left sub-tree*)

        GetNewNode := GetNewNode(root^.left,key)

    ELSE                   (* Into right sub-tree*)

        GetNewNode := GetNewNode(root^.right,key)

END;

(*--------------------------------------*)
```

```
PROCEDURE FillNode(node : TreeNodePtr;
                        key: KeyType);

(* This procedure initializes new node fields:
left and right pointers to NIL, the key to 'key'.
PARAMETERS: node --- pointer to the node,
            key --- the key.
*)

BEGIN

WITH node^ DO
    BEGIN
    left := NIL;
    right := NIL;
    nodeKey := key
    END

END;

(*-------------------------------------*)

PROCEDURE InsertKey(VAR root : TreeNodePtr;
                    key : KeyType);

(* This procedure inserts a key into a tree.
PARAMETERS: root --- pointer to tree root,
            key --- the key.
*)

BEGIN

FillNode(GetNewNode(root,key),key)

END;

(*-----------------------------------*)
```

```
FUNCTION BuildTree (VAR keyFile : KeyFileType) :
                                TreeNodePtr [PUBLIC];

(*This function builds a tree and returns
 a pointer to the tree.
PARAMETER: keyFile --- the file where the keys
           are.
RETURN: the function returns a pointer to the tree
        built.
*)

VAR
    key : KeyType;        (* holds current key. *)
    root : TreeNodePtr;   (*pointer to tree root.*)

BEGIN

root := NIL;

REPEAT          (*Loop reading keys and inserting*)
                (*them into the tree.            *)

    IF (EOF(keyFile)) THEN BREAK; (*Stop reading*)
                                  (*keys if reached*)
                                  (*  end of file. *)

    (* read a key and insert it into the tree.*)


    read (keyFile,key);
    InsertKey (root,key);

UNTIL key = SentinKey;

BuildTree := root

END;

(*-----------------------------------*)
```

```
PROCEDURE TraverseTreeInOrder(root: TreeNodePtr;
          PROCEDURE Action(key:KeyType)) [PUBLIC];

(* This procedure traverses a tree in order while
calling a procedure to process each key.
PARAMETERS: root --- pointer to tree root,
            Action --- procedure to process each
            key.
*)

BEGIN

IF root <> NIL
THEN
    BEGIN
    (* Traverse left sub-tree*)
    TraverseTreeInOrder(root^.left,Action);
    (* Process root key*)
    Action(root^.nodeKey);
    (* Traverse right sub-tree*)
    TraverseTreeInOrder(root^.right,Action)
    END

END;

(*--------------------------------------*)

PROCEDURE DisplayKey(key:KeyType) [PUBLIC];

(* This procedure displays a key on the screen.
PARAMETER: key --- key to display.
*)

BEGIN

write(key)

END;

END.
```

**INCLUDED DECLARATION FILE: TREE.DCL**

```
(* Declarations for the tree example *)

TYPE
    KeyType = CHAR;          (* Type of tree key*)
    KeyFileType = FILE OF KeyType;
    TreeNodePtr = ^TreeNode; (* Pointer to tree*)
    TreeNode = RECORD
                nodeKey : KeyType;
                left    : TreeNodePtr;
                right    : TreeNodePtr
               END;
```

**INSTRUCTIONS FOR COMPILING, LINKING, AND RUNNING EXAMPLE 3**

Example 3 is compiled exactly as Example 2, except that the two compilands are TreeMain.Pas and TreeModule.Pas. The file Tree.Dcl is included automatically in both files because the $INCLUDE metacommand is used in both source files.

After you have compiled, link the two object files, TreeMain.Obj and TreeModule.Obj, completing the Linker command form as shown below:

```
Link
  Object modules    TreeMain.Obj TreeModule.Obj
  Run file          TreeMain.Run
  [List file]
  [Publics?]
  [Line numbers?]
  [Stack size]
  [Max memory
      array size]
  [Min memory
      array size]
  [System build?]
  [Version]
  [Libraries]
  [DS Allocation]
  [Symbol file]
```

The resulting run file, TreeMain.Run, can be invoked by completing the Run command form as shown below. The parameter Inputfile is any file containing ASCII characters that you choose to use. Inputfile must be the name of a real file in your directory.

```
Run
  Run file          TreeMain.Run
  [Case]
  [Parameter 1]     Inputfile
  [Parameter 2]
  [Parameter 3]
  [Parameter 16]
```

**actual parameter.** See **formal parameter.**

**attribute.** An attribute gives additional information about a procedure or function. Attributes are available at the **extend level.** They are placed after the heading, enclosed in brackets, and separated by commas.

**body.** The body of a program or implementation is a list of statements enclosed with the reserved words BEGIN and END.

**compile time.** Compile time is the time when the compiler is executing, during which it compiles the source file and creates an object file.

**constant.** A constant is a value that is known before a program starts that will not change as the program progresses. A constant can be given an identifier, but you cannot alter the value associated with that identifier during the execution of the program. When you declare a constant, its identifier becomes a synonym for the constant itself.

**directive.** A directive gives information about the location of a procedure or function. A directive replaces the block of the procedure or function (declarations and body) and indicates that only the heading of the procedure or function occurs. Directives are available in standard Pascal.

**escape sequence.** An escape sequence is a sequence of characters that invokes special functions.

**expressions.** Expressions are constructions that evaluate to values.

**extend level.** The extend level is the language level that describes features specific to our version of Pascal, as opposed to standard Pascal.

**external reference.** An external reference is a variable or routine in one module that is referred to by a routine in another module. The variable or routine is often said to be "defined" or "public" in the module in which it resides.

The Linker tries to resolve external references by searching for the declaration of each such reference in other modules. If such a declaration is found, the module in which it resides is selected to be part of the executable module (if it is not already selected) and becomes part of your executable file. These other modules are usually library modules in the run-time library.

**field.** Each component of a record type is called a field. The definition of a record type specifies the type and an identifier for each field within the record.

**file.** A file is a structure that consists of a sequence of components, all of the same type.

**File Control Block (FCB).** There is a File Control Block for each open file. The FCB contains information about the file such as the device on which it is located, the user count (that is, how many file handles currently refer to this file), and the file mode (read or modify). The FCB is pointed to by a User Control Block and contains a pointer to a chain of File Area Blocks. The FCB is memory resident.

**formal parameter.** 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, value, or expression.

**function.** A function is a subprogram that can be invoked in expressions wherever values are called for. A function executes under the supervision of a main program. Functions can be nested within each other and can be called recursively.

A function can also be thought of as a procedure that returns a value of a particular type.

**heap.** The heap is a dynamically growing and shrinking region of memory allocated for pointer variables.

**identifier.** An identifier is a name that denotes the constants, variables, data types, procedures, functions, or other elements of a Pascal program.

**implementation of a unit.** The compilation unit where the actual code for a routine exists is called an implementation of the interface. Since the implementation does not define the routines it is implementing, the implementation must textually include the interface. Any compilation unit that wishes to call a routine implemented in a different compilation unit must then explicitly name the unit it wishes to use and also textually include the unit's interface. See also **interface** and **unit.**

**interface.** An interface contains the declarations for a routine. The code for the routine, however, exists in another compilation unit called the implementation of the interface. See also **unit.**

**link time.** Link time is the time when the Linker is executing, during which it links together object files and library files.

**M and N parameters.** M and N parameters are value parameters of type INTEGER and are used for formatting in various ways. M and N are expressions whose integer values are field-width parameters.

**metacommand.** A metacommand is a compiler directive that you use to control such options as optimization level, use of the source file during compilation, listing file format, and debugging and error handling.

**module.** Module is a general term for a discrete unit of code. There are several types of modules, including relocatable and executable modules. In addition in this version of Pascal, module is a specific type of Pascal compiland.

Modules are a less structured and less powerful way than units to combine several compilands into one program. A module is a program without a body. (Although there are no program statements in a module body, a module does become a procedure without parameters that can be called from other commands using the module identifier.)

**object module.** The object files created by the compiler are relocatable, that is they do not contain any absolute addresses. Linking produces an executable module, that is one that contains the necessary addresses to proceed with loading and running the program.

**operator.** An operator is a form of punctuation that indicates some operation to be performed, for example, the plus sign (+).

**ordinal type.** An ordinal type is a simple data type that is finite and countable. Ordinal types include the following simple types: INTEGER, WORD, CHAR, BOOLEAN, enumerated types, and subrange types.

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

**overlay.** An overlay is a code segment, made up of the code from one or more object modules. An overlay is loaded into memory from disk only when it is needed and is not permanently memory-resident.

**parameter.** A parameter provides a substitution mechanism that allows a process to be repeated with a variation of its arguments.

**pointer.** A pointer type is a set of values that point to variables of a given type and is used for creating, using, and destroying variables allocated from an area called the heap. Pointers are generally used for trees, graphs, and list processing. The use of pointers is portable, structured, and relatively safe.

**procedure.** A procedure is a subprogram that is invoked as a statement and executes under the supervision of a main program. Procedures can be nested within each other and can be called recursively.

**program.** A program is a series of instructions for the computer that perform a specified task. In Pascal, a program can include references to other compilable units, such as modules, or implementations of units, as well as the series of instructions themselves.

**record type.** A record structure acts as a template for conceptually related data of different types. The record type itself is a structure consisting of a fixed number of components, usually of different types.

**reference.** A reference to a variable or constant is an indirect way to access it.

**relocatable modules.** The module's code can be loaded and run at different locations in memory. Relocatable modules contain routines and variables represented as offsets relative to the start of the module. These routines and variables are said to be at relative offset addresses.

**routine.** A routine is code, residing in a module, that represents a particular procedure or function. More than one routine can reside in a module.

**run file.** A run file is a memory image of a task (in ready-to-run form) linked into the standard format required by the operating system loader.

**run time.** The time during which a compiled and linked program is executing. By convention, run time refers to the execution time of your program and not to the execution time of the compiler or the Linker.

**run-time library.** Contains the run-time routines needed to implement the Pascal language. A library module usually corresponds to a feature or subfeature of this version of Pascal.

**simple data type.** A simple data type is a data type that cannot be divided into other types. The simple data types fall into three categories: ordinal types, REAL, and INTEGER4.

**stack.** The stack is an area in the default data segment used for temporary storage of variables.

**statement.** A statement is a Pascal command that performs an action, such as computing, assigning, altering the flow of control, and reading and writing files. Statements denote actions that the program can execute. Statements are found in the bodies of programs, procedures, and functions and are executed as a program runs.

A simple statement has no parts that are themselves other statements; a structured statement consists of two or more other statements.

**string constant.** A string constant is a sequence of characters that can be expressions or values of the STRING type.

**string literal.** A string literal is a sequence of characters enclosed in single quotation marks.

**structured data type.** A structured data type is a data type that is composed of other types, for example, arrays.

**super type.** A super type is like a set of types or like a function that returns a type. A super type declaration determines the set of types that designators of that super type can assume; it also associates an identifier with the super type. Super type declarations also occur in the TYPE section. The only super types currently available are super arrays.

**tag field.** A record can have several variants, in which case a specific field called the tag field indicates which variant to use. The tag field can have an identifier and storage in the record.

**type.** A type is the set of values that a variable or value can have within a program. Types are either predeclared or declared explicitly. See also **super type.**

The type declaration associates an identifier with a type of value. You declare types in the TYPE section of a program, procedure, function, module, interface, or implementation (not in the heading of a procedure or function).

**unit.** A unit is a program module made up of an interface and an implementation. A unit is a group of procedures and functions (but no main program) that are compiled together. A unit has two parts, an implementation and an interface.

The unit interface specifies all identifiers that are defined in the unit that have definitions used in another compilation unit. The unit implementation contains the actual code for all procedures and functions carried out by the unit.

The unit is preceded by the keyword UNIT for a unit provided, and with the keyword USES for a unit required.

**value.** A value can be any of the following: a variable, a constant, a function designator, a component of a value, or a variable referenced by a reference value.

**value section.** You use the VALUE section to give initial values to variables in a program, module, procedure, or function. You can also initialize the variable in an implementation, but not in an interface.

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

**This index covers both Volumes 1 and 2. Sections 1 through 12 are in Volume 1. Sections 13 through the Glossary are in Volume 2.**

**Page numbers in boldface indicate the principal discussion of a topic.**

# USER'S COMMENT SHEET

Pascal Reference Manual, Volume 2
Third Edition
A-09-00868-01-A

---

*We welcome your comments and suggestions. They help us improve our manuals. Please give specific page and paragraph references whenever possible.*

*Does this manual provide the information you need? Is it at the right level? What other types of manuals are needed?*

*Is this manual written clearly? What is unclear?*

*Is the format of this manual convenient in arrangement, in size?*

*Is this manual accurate? What is inaccurate?*

Name _____ Date _____

Title _____ Phone _____

Company Name/Department _____

Address _____

City _____ State _____ Zip Code _____

*Thank you. All comments become the property of Convergent Technologies, Inc.*

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

# BUSINESS REPLY MAIL

FIRST CLASS MAIL PERMIT NO. 1807 SAN JOSE, CA

POSTAGE WILL BE PAID BY ADDRESSEE

**Convergent Technologies**
**Attn:** Technical Publications
2700 North First Street
PO Box 6685
San Jose, CA 95150-6685

Fold Here

Please Do Not Staple

Tape

Convergent

2700 North First Street
San Jose, CA 95150–6685

*Printed in USA*