MicroPower/Pascal Language Guide

Order No. AA-M389E-TK

digital

software

MicroPower/Pascal Language Guide

Order No. AA–M389E–TK

June 1987

This manual describes the elements of the MicroPower/Pascal Language and its real-time extensions. The manual is intended as a reference manual for use in preparing MicroPower/Pascal programs.

Revision/Update Information:	This manual supersedes the *MicroPower/Pascal Language Guide*, AA–M389D–TK.
Operating System and Version:	Micro/RSX Version 3.0 RSX–11M Version 4.2 RSX–11M–PLUS Version 3.0 RT–11 Version 5.2 VAX/VMS Version 4.0
Software Version:	MicroPower/Pascal–Micro/RSX Version 2.4 MicroPower/Pascal–RSX Version 2.4 MicroPower/Pascal–RT Version 2.4 MicroPower/Pascal–VMS Version 2.4

Digital Equipment Corporation   Maynard, Massachusetts

First Printing, February 1982
Updated, June 1982
Revised, October 1982
Updated, February 1983
Revised, July 1983
Updated, February 1984
Revised, June 1985
Updated, April 1986
Updated, October 1986
Revised, June 1987

This document was prepared using an in-house documentation production system. All page composition and make-up was performed by TEX, the typesetting system developed by Donald E. Knuth at Stanford University. TEX is a trademark of the American Mathematical Society.

Contents

Chapter 2 Data Types

Chapter 3 Expressions

Chapter 12 Process Management Requests

Chapter 13 Binary and Counting Semaphore and Mutex Management Requests

Chapter 14 Queue Semaphore Management Requests

Chapter 15 Ring Buffer Management Requests

Chapter 20 Miscellaneous Requests

Preface

This manual describes the MicroPower/Pascal language, which is an extension of the standard Pascal programming language proposed by the International Standardization Organization (ISO).

Structure of This Document

This 2-part manual contains 20 chapters and 9 appendixes.

Part I describes the standard Pascal language, as defined by Niklaus Wirth, and a number of DIGITAL-created extensions.

- Chapter 1 provides an overview of the MicroPower/Pascal language and describes the structure of a MicroPower/Pascal program.
- Chapter 2 presents detailed information on standard data types.
- Chapter 3 discusses expressions involving constants, variables, function designators, and operators.
- Chapter 4 describes the statements of the declaration section.
- Chapter 5 describes the statements that perform the actions of a program, including the process invocation (creation) and procedure call statements.
- Chapter 6 describes the declaration and structure of subprograms (procedures, functions, and processes).
- Chapter 7 describes compilation units and independent compilation.
- Chapter 8 describes the predeclared data manipulation functions and procedures supplied with the MicroPower/Pascal software.
- Chapter 9 describes the syntax and use of the predeclared input/output procedures. Those procedures provide access to standard I/O devices and the MicroPower/Pascal file system.
- Chapter 10 describes the syntax and use of the MicroPower/Pascal attributes. A syntactic feature of the MicroPower/Pascal software, attributes provide additional control over the properties of a variety of language elements and constructs.

Part II describes the real-time programming requests (predeclared programming procedures and functions) that extend the capabilities of standard (sequential) Pascal to allow access to the real-time concurrent programming services of the MicroPower/Pascal kernel.

- Chapter 11 discusses the conventions to follow when using the real-time programming requests.
- Chapter 12 describes process management requests, which provide for dynamic process deletion, suspension, resumption, status reporting, and forced termination.
- Chapter 13 describes semaphore management requests, which provide the basic synchronization mechanisms between cooperating processes through signal and wait operations on binary and counting semaphores.
- Chapter 14 describes queue semaphore management requests, which provide interprocess message transmission service by combining message packet queuing and dequeuing with semaphore signal and wait operations.
- Chapter 15 describes ring buffer management requests, which provide for variable-length data transfers without the need for explicit synchronization between sending and receiving processes.
- Chapter 16 describes interrupt management requests, which provide interrupt dispatching control for processes that manage an I/O or a clock device.
- Chapter 17 describes exception condition management requests, which direct the dispatching of hardware and software exception conditions to an appropriate exception-handling procedure or exception-handling service routine. The requests also permit a process to report a software exception.
- Chapter 18 describes the memory allocation and mapping requests, which allow a process to control dynamically the allocation of target memory.
- Chapter 19 describes the timer requests, which allow a process to set and obtain the kernel-maintained system time and to implement timed process blocking.
- Chapter 20 describes a miscellaneous group of requests that permit a process to obtain heap storage space information, create a logical name that translates to a specified string value, eliminate the translation value defined for a given logical name, obtain build-time hardware configuration information, detect a power failure, and obtain the translation string defined for a given logical name.
- Appendix A lists the ASCII character set.
- Appendix B provides a complete summary of MicroPower/Pascal language syntax, using a modified version of the Backus-Naur form.
- Appendix C explains the use of compile-time aids for the programmer.
- Appendix D shows the declarations of the predeclared data types used by the real-time programming requests.
- Appendix E describes the rules by which the compiler allocates storage of the standard data types.
- Appendix F lists the MicroPower/Pascal attributes and the entities to which they apply.

- Appendix G lists the identifiers that are predefined in the MicroPower/Pascal language as the names of files, functions, procedures, types, and values.
- Appendix H describes the limitations of the MicroPower/Pascal compiler and suggests programming techniques to avoid encountering them.
- Appendix I lists the system %INCLUDE and module files and the predefined MicroPower/Pascal I/O and real-time requests that are defined in them.

Intended Audience

This manual is intended for readers who know the Pascal language and who are familiar with concepts of concurrent programming. This manual is not a tutorial manual but instead is primarily for reference. If either Pascal or concurrent programming concepts are new to you, see the bibliography at the end of this manual for a partial list of applicable tutorial manuals.

For detailed information about the MicroPower/Pascal run-time system and utility programs, see the Associated Documents section.

Conventions Used in This Document

Convention	Meaning
{ }	Braces enclose lists from which you must choose one item. For example: $\left\{\begin{array}{l}\text{expr}\\ \text{statement}\end{array}\right\}$
...	Horizontal ellipses indicate that the preceding item can be repeated. For example: filename, ...
{ },...	Braces followed by a comma or a semicolon and horizontal ellipses mean that you can repeat the enclosed item one or more times, separating the items with commas or semicolons as applicable. For example: REPEAT { statement } ;...
⟦ ⟧	Double brackets in the statement format descriptions enclose items that are optional. For example: ⟦PACKED⟧
[]	Single brackets are used in the statement syntax for arrays, sets, and attributes. For example: ARRAY [subscript1]
(* *) { }	Parentheses and asterisks or single braces in the same typeface as the text are part of the statement syntax. This notation is used in program comment text. For example: (* This is a comment. *)

Convention	Meaning

ITEMS IN UPPERCASE CHARACTERS	Uppercase characters in syntax descriptions indicate MicroPower/Pascal predefined identifiers and reserved words that you must not abbreviate. For example:
	BEGIN
	END

items in lowercase characters	Lowercase characters in format descriptions represent elements that you must replace according to the description in the text.
. . .	Vertical ellipses indicate that one or more statements in a figure or an example are not shown.
numbers	Unless otherwise specified, all numbers are in decimal radix. For example: 3452 76311(octal)
standard Pascal	The phrase "standard Pascal" refers to the Pascal language as described by the International Standards Organization in the *ISO Specification for Computer Programming Language PASCAL,* Draft Proposal 7185, Level 0.

Associated Documents

The following software documentation is required for complete reference purposes. Refer to the documentation list for your host operating system.

- **RT-11 Host:**

  MicroPower/Pascal-RT documentation set. A complete list of documents is contained in the *MicroPower/Pascal-RT Documentation Directory.*

  RT-11 host operating system documentation set. A subset of the RT-11 V5.2 documentation set is contained in the MicroPower/Pascal-RT documentation set. No additional RT-11 documentation is required for MicroPower/Pascal-RT application software development.

- **RSX-11M/M-PLUS Host:**

  MicroPower/Pascal-RSX documentation set. A complete list of documents is contained in the *MicroPower/Pascal-RSX Installation Guide.*

  RSX-11M/M-PLUS host operating system documentation set. Refer to the documentation set supplied with your host operating system.

- **VAX/VMS Host:**

  MicroPower/Pascal-VMS documentation set. A complete list of documents is contained in the *MicroPower/Pascal-VMS Installation Guide.*

  VAX/VMS host operating system documentation set. Refer to the documentation set supplied with your host operating system.

Chapter 1 Introduction

The MicroPower/Pascal language, a combination of the standard Pascal language and a programmable run-time system, was developed for use under the RT–11, Micro/RSX–11, and VMS operating systems to produce microprocessor code appropriate for storage in read-only memory. The MicroPower/Pascal language includes all the standard Pascal language elements plus the following extensions:

- Attributes on data objects and the names of functions, modules, procedures, processes, and programs
- Binary, hexadecimal, and octal integers
- Compile-time inclusion of external text files into source program text
- Concurrent processes programmed in Pascal that control run-time procedures by using a combination of predeclared and user-written routines
- Dollar sign ($) and underscore (_) characters in identifiers
- Error-handling and interrupt capabilities
- Extended parameter specifications to support standard PDP–11 calling sequence
- External functions, procedures, processes, and variables
- LONG_INTEGER data type (32 bits)
- MODULE declaration to support separate and independent compilation
- Nonpositional parameters and default parameter values
- OTHERWISE clause in the CASE statement
- Predeclared procedures and functions that facilitate the creation of multiprogrammed applications
- Structured constants
- Type cast operator that allows selective overriding of type checking
- UNSIGNED data type

This chapter presents an overview of the important concepts in Pascal and illustrates the structure of a Pascal program. The chapter also describes Pascal's lexical elements: the character set, identifiers, reserved words, and special symbols. The final section explains how to document a program.

1.1 Terminology

A Pascal program performs operations on data items known as constants, variables, and function identifiers. A constant is a quantity with an unchanging value. A constant to which you give a name is called a symbolic constant. A variable is a named entity for storing a program's modifiable data. A function identifier initiates execution of a group of statements with which it is associated and returns a value. The data type of the function identifier is the type of the value it returns.

1.1.1 Data Types

Every data item has an associated data type. A data type, usually indicated by a type identifier, determines both the range of values a data item can assume and the operations that can be performed on it. In addition, the type implicitly indicates how much storage space is required for the data item's possible values.

Pascal provides identifiers for many predefined types. Thus, a program's operations can involve arrays, Boolean and character data, integers, pointers to dynamic variables, real numbers, records, sets, and unsigned values. Pascal also allows you to create your own types by defining an identifier of your choice to represent a list of objects or a range of values.

The type of a constant is the type of its corresponding value. The types of a variable and a function identifier are established when they are declared and cannot be changed. Although variables and function identifiers can change in value any number of times, the values they assume must be within the range established by the type. A variable does not assume a value until the program assigns it one. A function identifier is assigned a value during the execution of the function.

Pascal associates types with both data items and expressions. An expression is the computation of a value resulting from a combination of constants, function designators, operators, and variables. You can form expressions by using arithmetic, logical, relational, set, and string operators. Arithmetic expressions produce integer, long integer, real-numbered, or unsigned values. Logical, relational, string, and most set expressions yield Boolean results. Other set expressions form the differences, intersection, and union of two sets.

1.1.2 Definitions and Declarations

Pascal requires you to define every constant and user-created type and to declare every function, label, procedure, process, and variable used in your program. The declaration section of the program contains CONST, FUNCTION, LABEL, PROCEDURE, PROCESS, TYPE, and VAR sections, in which you define and declare the data and subprograms that your program uses. All except the LABEL section introduce identifiers and indicate what they represent. A LABEL section declares numeric labels that correspond to executable statements accessed by the GOTO statement.

1.1.3 Executable Statements

The executable section of a program contains the statements that specify the program's actions. The executable section is delimited by the reserved words BEGIN and END. Between BEGIN and END are conditional and repetitive statements, statements that assign values to variables and function identifiers, and statements that control program execution.

1.1.4 Subprograms

The MicroPower/Pascal language allows you to group declarations, definitions, and executable statements into an identifiable entity called a subprogram. The two categories of subprograms are routines and processes.

1.1.4.1 Routines

Routines may be declared as either procedures or functions and are a convenient way to isolate the individual tasks that the program is to accomplish. Procedures are usually written to perform a series of actions, whereas functions are written to compute a value. Routines do not exist independently of the program; they are called either by an executable statement known as a procedure call or by a function identifier appearing within an expression. Routines always execute to completion before returning control to the caller. The MicroPower/Pascal software supplies many predeclared routines that perform commonly used operations, including input, output, and requests for real-time services from the MicroPower/Pascal kernel.

A routine consists of a heading and a block. The heading provides the routine's name, usually a list of formal parameters that declare the input data for the routine, and, in the case of functions, the type of the result. The block consists of an optional declaration section and an executable section. When the declaration section is present, it declares data that is local to the routine—data that is unavailable outside the routine.

Pascal is a block-structured language, allowing you to nest routine blocks not only within the program but also within other routines. Each routine can make its own local definitions and declarations and can even redeclare an identifier that has been declared in a block at an outer nesting level. A routine declared at an inner nesting level has access to the declarations and definitions made in all blocks that enclose it.

1.1.4.2 Processes

MicroPower/Pascal processes are similar in structure to procedures, but differ in purpose in that they execute concurrently—logically in parallel—with the execution of other processes in a given program. The process construct lets you decompose an otherwise monolithic, sequential program into autonomous subprograms that are scheduled for independent execution and that can be triggered by appropriate events. The process construct provides a simpler conceptual approach than one that uses sequential programming techniques to solving real-time problems. The real-time programming requests, described in Part Two, provide the process synchronization and communication services necessary for effective concurrent programming. The *MicroPower/Pascal Run-Time Services Manual* describes processes and concurrent programming.

1.1.5 Compilation Units

The two kinds of compilation units that MicroPower/Pascal provides are the program and the module. Although the structures of the two are similar, programs have executable blocks at the outermost nesting level, whereas modules do not. A program can be compiled, built, and executed by itself with only the system modules that are included automatically.

A program consists of a heading and a block, just as a routine does. The heading consists of the program's name and possibly a list of identifiers. The declaration section of the program's block declares data that is accessible at all program levels, including all nested routines.

A module consists of a heading, the format of which is similar to the program heading, and a declaration section. A module does not contain any executable sections at the outermost program level and though a module can be compiled separately from a program, a module cannot be executed unless it is merged with a program.

Chapter 7 specifies the syntax for both programs and modules.

1.1.6 Attributes

For systems-programming applications, standard Pascal does not allow sufficient control over certain aspects of a program. By including a class of language extensions known as attributes, the MicroPower/Pascal language allows you to control program elements for which the compiler otherwise provides defaults. Some examples of program elements controlled by attributes are the addressing boundaries on which data items are aligned, default allocation sizes for data types, and the form of storage a variable occupies. According to the needs of your program, you can change the defaults by associating attributes with compilation units, formal parameters, routines, and variables.

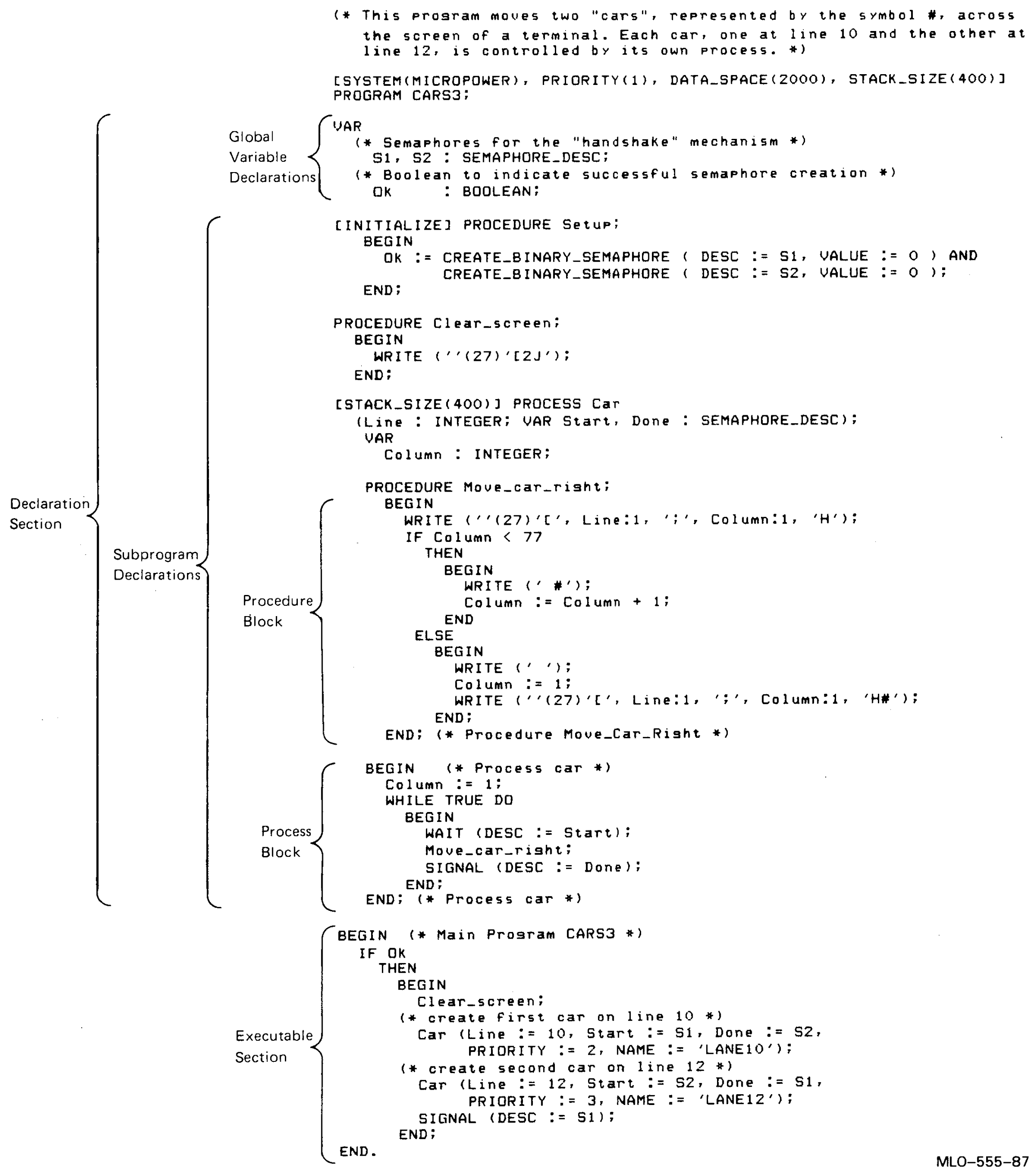
The syntax for specifying attributes is given throughout this manual in the sections describing type definitions, variable declarations, and routine, program, and module headings.

Chapter 10 discusses explanations, defaults, and rules for the attributes.

1.1.7 Structure of a Pascal Program

Figure 1–1 illustrates some of the parts of a typical MicroPower/Pascal program.

Figure 1-1: Structure of a MicroPower/Pascal Program



```
(* This program moves two "cars", represented by the symbol #, across
   the screen of a terminal. Each car, one at line 10 and the other at
   line 12, is controlled by its own process. *)

[SYSTEM(MICROPOWER), PRIORITY(1), DATA_SPACE(2000), STACK_SIZE(400)]
PROGRAM CARS3;

VAR
   (* Semaphores for the "handshake" mechanism *)
    S1, S2 : SEMAPHORE_DESC;
   (* Boolean to indicate successful semaphore creation *)
    OK      : BOOLEAN;

[INITIALIZE] PROCEDURE Setup;
   BEGIN
     OK := CREATE_BINARY_SEMAPHORE ( DESC := S1, VALUE := 0 ) AND
           CREATE_BINARY_SEMAPHORE ( DESC := S2, VALUE := 0 );
   END;

PROCEDURE Clear_screen;
  BEGIN
    WRITE (''(27)'[2J');
  END;

[STACK_SIZE(400)] PROCESS Car
  (Line : INTEGER; VAR Start, Done : SEMAPHORE_DESC);
   VAR
     Column : INTEGER;

   PROCEDURE Move_car_right;
     BEGIN
       WRITE (''(27)'[', Line:1, ';', Column:1, 'H');
       IF Column < 77
         THEN
           BEGIN
             WRITE (' #');
             Column := Column + 1;
           END
        ELSE
          BEGIN
            WRITE (' ');
            Column := 1;
            WRITE (''(27)'[', Line:1, ';', Column:1, 'H#');
          END;
     END; (* Procedure Move_Car_Right *)

   BEGIN   (* Process car *)
     Column := 1;
     WHILE TRUE DO
       BEGIN
         WAIT (DESC := Start);
         Move_car_right;
         SIGNAL (DESC := Done);
       END;
   END; (* Process car *)

BEGIN  (* Main Program CARS3 *)
  IF OK
    THEN
      BEGIN
        Clear_screen;
       (* create first car on line 10 *)
         Car (Line := 10, Start := S1, Done := S2,
              PRIORITY := 2, NAME := 'LANE10');
       (* create second car on line 12 *)
         Car (Line := 12, Start := S2, Done := S1,
              PRIORITY := 3, NAME := 'LANE12');
         SIGNAL (DESC := S1);
       END;
END.
```

1.2 Elements of the Language

A program is composed of lexical elements: individual symbols, such as arithmetic operators, or words that have special meanings in Pascal. The basic unit of any lexical element is a character, which must be a member of the ASCII character set, as described in Section 1.2.1. Some characters are special symbols that Pascal uses as statement delimiters, operators, and elements of the language syntax. Special symbols are presented in Section 1.2.2.

The words that Pascal uses are combinations of alphabetic characters, dollar signs, percent signs, and underscores. Pascal reserves some words for the names of executable statements, operations, and some of the predefined data types. Reserved words are listed in Section 1.2.3. Other words in a Pascal program are called identifiers. Predeclared identifiers represent routines and data types provided by Pascal. Other identifiers are user declared to name constants, programs, variables, and any other necessary program segment that is not already named. Section 1.2.4 explains the use of both kinds of identifiers.

1.2.1 Character Set

Pascal uses an extended American Standard Code for Information Interchange (ASCII) character set (see Appendix A). This extended ASCII character set contains 256 characters in the following categories:

- The uppercase and lowercase letters A to Z
- The numbers 0 to 9
- Special characters, such as the ampersand (&), equal sign (=), and question mark (?)
- Nonprinting characters, such as the bell, carriage return, line feed, space, and tab
- Extended, unspecified characters with numeric codes from 128 to 255

The MicroPower/Pascal compiler does not distinguish between uppercase and lowercase letters except in character and string constants. For example, the word PROGRAM has the same meaning when written as any of the following:

```
PROGRAM
PRogrAm
program
```

The following constants, however, represent different characters:

```
'b'
'B'
```

The following string constants are also different:

```
'BREAD AND ROSES'
'Bread and Roses'
```

1.2.2 Special Symbols

Pascal uses special symbols to represent delimiters, operators (arithmetic, logical, relational, set, and string), and other syntax elements. The special symbols are listed below.

Name	Symbol	Name	Symbol
Assignment operator	:=	Minus sign	–
Brackets	[] or (. .)	Multiplication	*
Colon	:	Not equal	<>
Comma	,	Parentheses	()
Comment delimiters	(* *) or { }	Decimal point or Field selector	.
Division	/	Plus sign	+
Equal	=	Pointer	^ or @
Greater than	>	Semicolon	;
Greater than or equal to	> =	Subrange operator	..
Less than	<	Type cast operator	::
Less than or equal to	<=		

1.2.3 Reserved Words

The MicroPower/Pascal language reserves the words listed below as names for statements and operators. Reserved words shown in **bold** type are extensions to standard Pascal. This manual shows reserved words in uppercase. You can use reserved words in your program only in the contexts in which Pascal allows them. You cannot redefine a reserved word for use as an identifier.

AND	DOWNTO	FUNCTION	NIL	**PROCESS**	TO
ARRAY	ELSE	GOTO	NOT	PROGRAM	TYPE
BEGIN	END	IF	OF	RECORD	UNTIL
CASE	**EXTERNAL**	IN	OR	REPEAT	VAR
CONST	FILE	LABEL	**OTHERWISE**	**SEQ11**	WHILE
DIV	FOR	MOD	PACKED	SET	WITH
DO	**FORWARD**	**MODULE**	PROCEDURE	THEN	

1.2.4 Identifiers

Pascal uses identifiers to name constants, formal parameters, functions, modules, procedures, processes, programs, record fields, types, and variables. An identifier is a sequence of digits, dollar signs ($), letters, and underscores (_), with the following restrictions:

- An identifier cannot start with a digit.
- An identifier must be unique in the first 31 characters within the block in which it is declared.
- An identifier must not contain any spaces or special symbols.

Pascal scans only the first 31 characters of an identifier for uniqueness; the remaining characters are ignored. Thus, the compiler will ignore the second declaration of a pair of identifiers that are not unique within 31 characters.

The following examples are valid and invalid identifiers:

Valid

```
FOR2N8
MAX_WORDS
UPTO
LOGICAL_NAME_TABLE                   unique in first
LOGICAL_NAME_SCANNER                 31 characters
SYS$CREMBX
```

Invalid

```
4AWHILE                              starts with a digit
UP&TO                                contains an ampersand
YEAR_END_80_MASTER_FILE_TOTAL_DISCOUNT not unique in first
YEAR_END_80_MASTER_FILE_TOTAL_DOLLARS  31 characters --
                                       duplicate is ignored
```

Note

Although the MicroPower/Pascal language allows the dollar sign ($) in identifiers, this character has a special meaning to the MicroPower/Pascal system software in some contexts. You should restrict the use of the dollar sign ($) to identifiers representing DIGITAL-supplied symbolic names.

1.2.4.1 Predefined Identifiers

The identifiers listed in Appendix G are predefined within the MicroPower/Pascal language as names of files, functions, procedures, types, and values. This manual shows predefined identifiers in uppercase.

You can change a predefined identifier to denote another item. If you do so, however, you can no longer use the identifier for its usual purpose within the scope of the block in which it is redefined. See Section 6.4.1 for a description of the scope of identifiers. For example, the identifier READ denotes the READ procedure, which performs input operations. If you use the word READ to denote something else (say, a variable) you cannot use the READ procedure within the same block. Because you could lose access to a useful language feature, you should avoid redefining predeclared identifiers.

1.2.4.2 User-Defined Identifiers

These identifiers denote the names of constants, record fields, formal parameters, functions, modules, procedures, processes, programs, user-defined types, and variables. User-defined identifiers represent significant data actions, structures, and values that are not represented by a predeclared identifier, reserved word, or special symbol.

1.3 Documenting Your Program

You can insert lines of comment text in your program to document its operation by enclosing the text within matching pairs of comment delimiters. You can place a comment anywhere that a space is valid.

Syntax

$$\left\{ \begin{matrix} \{ \\ (* \end{matrix} \right\} \text{ comment-text } \left\{ \begin{matrix} \} \\ *) \end{matrix} \right\}$$

comment-text

Any ASCII character other than a comment delimiter. The comment text must be enclosed in the delimiters that you choose.

Example

```
{ This is a comment. }

(* This is a comment too. *)
```

Chapter 2
Data Types

In Pascal, data types are divided into scalar, structured, and pointer categories. Scalar data types are the building blocks for the structured types. The pointer data type lets you refer to a variable indirectly. A scalar type is an ordered group of values. A value of a particular scalar type is always greater than, equal to, or less than another value of the same type. For example, the scalar type INTEGER denotes positive and negative integers. The integers follow a certain order; for example, -700 is less than 2.

MicroPower/Pascal supplies predefined scalar types for integer, unsigned integer, character, Boolean, and real data. In addition, you can define other scalar types to fit your needs. You construct your own scalar type either by enumerating each value of the type or by defining a subrange of another scalar type other than the real types. Values of user-defined scalar types are ordered like those of predefined scalar types.

The set of scalar data types is divided into subsets of ordinal types and real types. The ordinal types are INTEGER, LONG_INTEGER, UNSIGNED, CHAR, BOOLEAN, and user-defined enumerated and subrange types. Section 2.1 discusses the ordinal types; Section 2.2 discusses the type REAL.

Pascal has four structured types: records, arrays, sets, and files. Structured data types let you process groups of scalar, structured, or pointer data items. For example, you could have an array of integers, an array of arrays, a record of integers and characters, a file of records, or a set of an enumerated type. Sections 2.3 to 2.8 describe the predefined structured types.

Pascal lets you pack structured data types to save storage space. Packed structures are stored in as few bits as is feasible. To create a packed structured type, specify the modifier PACKED in the applicable type definition (see Chapter 4). You can determine the storage allocation for packed and unpacked structures with the information provided in Appendix E.

Type compatibility rules determine the operations and assignments you can perform with data items of different types. The complete rules of type compatibility are presented in Section 2.9.

You specify the types of data that your program will use by including statements in the declaration section (see Chapter 4). See Section H.3 for a discussion of compiler limitations governing the placement of type definitions in a program.

2.1 Ordinal Data Types

The ordinal data types let you specify entities that have an ordinal sequence based on a one-to-one correspondence with the set of integers. The components of an ordinal data type are ordered so each has a unique ordinal value indicating its position in a list of all the values of that type. The ordinal data types are discussed individually in the following subsections.

Three predeclared functions operate only on expressions of ordinal types and return information about the type's ordered sequence of values. Each value (except the smallest) of an ordinal type has a predecessor, which you can determine by using the PRED function. Similarly, each value (except the largest) of an ordinal type has a successor, which you can determine by using the SUCC function. The ORD function finds the ordinal value of any expression of an ordinal type and returns that value as an integer. The ordinal value of an integer is the integer itself.

Examples

```
1. ORD(23) (* is 23 *)

2. ORD(-1984) (* is -1984 *)

3. TYPE COLOR = (red,white,blue);

   ORD (white) (* is 1 *)

   PRED (blue) (* is white *)

   SUCC (white) (* is blue *)
```

2.1.1 INTEGER and LONG_INTEGER

Variables of type INTEGER and LONG_INTEGER can assume both positive and negative integer values. Type INTEGER values can be from –32768 to 32767. The value 32767 is known by the predefined constant identifier MAXINT. Type LONG_INTEGER values can be from –2147483647 to 2147483647.

You indicate a decimal integer constant by using decimal digits and the optional plus and minus signs.

Examples

A minus sign (–) must precede a negative integer value. A plus sign (+) may precede a positive integer but is not required. No commas or decimal points are allowed.

```
     17
   -333
      0
     +1
  21343
9826492
```

In addition to decimal notation, MicroPower/Pascal software lets you specify integer constants in binary, hexadecimal, and octal notations. You can use constants written in those notations wherever decimal integer constants are permitted.

To specify an integer constant in binary, hexadecimal, or octal notation, place a percent sign (%) and a letter in front of the number and apostrophes around the number. The appropriate letters, which may be either uppercase or lowercase, are B for binary notation, X for hexadecimal notation, and O for octal notation. An optional plus or minus sign may precede the percent sign to indicate a positive or a negative value. Intervening spaces are ignored.

Examples

```
-%B'110001'
%b'10 00 0011'
%O'112 101 103 113'
%O'7712'
-%o'473'
+%X'53 A 1'
-%x'DEC'
```

2.1.2 UNSIGNED

The UNSIGNED data type identifies integer values from 0 through 65535. The range of unsigned values includes no negative numbers. UNSIGNED is a machine-dependent type that identifies the range of positive numbers representable in one 16-bit word.

When a program contains an integer constant greater than MAXINT, that constant is treated as having type UNSIGNED or LONG_INTEGER if its value cannot be represented by 16 bits of binary data. Unsigned constants can use binary, decimal, hexadecimal, and octal notations, as described for integers in Section 2.1.1. Integer constants not greater than MAXINT are always treated as having type INTEGER. The ORD function, described in Section 8.18, converts unsigned values to integer values just as it converts other ordinal types to integers.

2.1.3 CHAR

A value of type CHAR is a single character from the ASCII character set, as listed in Appendix A. To specify a character value, enclose an ASCII character in apostrophes. The apostrophe character must be typed twice within apostrophes.

Examples

```
'A'
'Z'
'0'
'.'
''''
'?'
```

You must represent strings of characters, such as 'CONEHEAD' and '****', as packed arrays of characters (see Section 2.4.2). When you use the ORD function on an expression of type CHAR, the function returns the ordinal value in the ASCII character set of the character value stored in the variable.

Example

```
ORD(Q_Character);
```

Suppose that the variable Q_Character is of type CHAR and has a value of 'Q'. The expression returns 81, the ordinal value of uppercase Q in the ASCII character set.

The order of the ASCII character set may not be what you expect if you have not used it before. Although the numeric characters are in numerical order and the alphabetic characters are in alphabetical order, all uppercase characters have lower ordinal values than all lowercase characters.

Examples

1. `ORD('0') is less than ORD('9')`
2. `ORD('A') is less than ORD('Z')`
3. `ORD('Z') is less than ORD('a')`

2.1.4 BOOLEAN

The BOOLEAN data type consists of the values FALSE and TRUE. Pascal defines those values as predeclared identifiers and orders them so FALSE is less than TRUE. Thus, the ORD function applied to the Boolean value FALSE returns the integer 0; if the value is TRUE, ORD(TRUE) returns the integer 1. Boolean values are the result of testing relationships for truth or validity.

2.1.5 Enumerated Types

An enumerated type is an ordered set of values denoted by identifiers. To define an enumerated type, list in order all the identifiers denoting the constant values of the type; then enclose the list in parentheses.

Syntax

({ identifier } ,...)

identifier

A constant value of the type.

The values of an enumerated type follow a left-to-right order. Thus, any identifier in the list is greater than all identifiers to its left and less than all identifiers to its right.

Example

```
TYPE
  Seasons=(Spring, Summer, Fall, Winter);
BEGIN
  IF Spring < Fall
  THEN WRITELN ('TRUE')
END.
```

The relational expression above is TRUE, because Spring precedes Fall in the list of constant values.

The enumerated type definition associates an ordinal value with each value in the type. The ordinal value of the first identifier is 0; the ordinal value of the second identifier is 1, and so forth. You can apply the ORD function to values of enumerated types. For example, with the enumerated type Seasons, the function ORD(Summer) is valid and returns the integer 1, because Summer is the second value listed.

The only restriction on the values of an enumerated type is that a value identifier cannot be defined for any other purpose within the current scope.

Example

Suppose that you have the following enumerated type:

```
TYPE
  Some_Seasons=(Fall, Winter, Spring);
```

That enumerated type cannot be defined in the same program as the type shown in the previous example, because the identifiers Spring, Fall, and Winter would not be unique.

The following examples are enumerated types that you could define:

```
TYPE
  Drinks=(Milk, Water, Cola, Beer);
  Activities=(Swim, Run, Ski);
  Cookies=(Oatmeal, Sugar, Peanut_Butter, Choc_Chip);
```

2.1.6 Subrange Types

A subrange specifies a limited portion of another ordinal type (called the parent type) for use as a type.

Syntax

lower-limit..upper-limit

lower-limit
The constant value of the lower limit of the subrange.

upper-limit
The constant value of the upper limit of the subrange.

The subrange type is defined only for the values between and including the lower and upper limits. The limits you specify must be single values of the parent type. The value of the upper limit must be greater than or equal to the value of the lower limit. The subrange symbol (..) separates the limits of the subrange.

The parent type can be any enumerated or predefined ordinal type. The values in the subrange are in the same order as in the parent type. You can therefore use the ORD function with an identifier of a subrange type; the result is the ordinal value of the identifier with respect to the parent type. In general, you can use a subrange type anywhere in the program that its parent type is legal. The rules for operations and value initializations on a subrange are the same as for its parent type. A value of a subrange type is converted to a value of its parent type before it is used in an operation.

The use of subrange types can make a program clearer. For example, integer values for the days of the year range from 1 to 366. Any value outside that range is incorrect. You could specify an integer subrange for the days of the year as 1..366 and give it the variable identifier Day_Of_Year. By specifying a subrange, you indicate that the values of the variable Day_Of_Year are restricted to the integers from 1 to 366. If you use the CHECK option at compile time, the system generates a run-time error for an out-of-range assignment to a subrange variable. In this example, such an error occurs when an integer less than 1 or greater than 366 is assigned to Day_Of_Year.

Examples

The following examples are subrange types that you could create:

```
TYPE
  Year=(Jan, Feb, Mar, Apr, May, Jun, July, Aug, Sept, Oct, Nov, Dec);
  Single_Digits = '0'..'9';         (* Single-digit characters *)
  Alpha_First_Half = 'A'..'M';      (* First half of alphabet *)
  Month_Days = 1..31;               (* The days of the month *)
  Year_First_Half = Jan..Jun;       (* First half of year *)
  Year_Second_Half = Jul..Dec;      (* Second half of year *)
```

2.2 REAL Types

The identifier REAL denotes the real number type. Numbers of type REAL have the following range of values and degree of precision:

Smallest negative value:	$-2.938736 \times 10^{-39}$
Largest negative value:	-1.701411×10^{38}
Smallest positive value:	2.938736×10^{-39}
Largest positive value:	1.701411×10^{38}
Precision:	7 decimal digits (approximately)

Real number values can be written in either fixed-point decimal or exponential notation. Real numbers in fixed-point notation use the set of decimal digits and a decimal point, as well as an optional plus or minus sign.

Examples

The following are valid real number values in fixed-point notation:

```
  2.4
893.2497
 -0.01
  8.0
-23.18
  0.0
```

In fixed-point notation, at least one digit must appear on each side of the decimal point. That is, a 0 must always precede the decimal point of a number between 1 and -1, and a 0 must follow the decimal point of a whole number.

Some numbers are too large or too small to be written conveniently in the format above. Pascal provides exponential, or floating-point, notation as a second way of writing real numbers. Real numbers in exponential notation use an optional plus or minus sign, a real number or an integer number, an uppercase or lowercase letter E, and an integer exponent with its optional plus or minus sign. The letter E after the value indicates that the value is to be multiplied by a power of 10. The integer following the letter E shows which power of 10; positive or negative.

Examples

```
 2.3E2
-0.07e4
10.0E-1
-201E+3
-3.14159E0
```

The real number 237.0 can be represented in the following ways:

```
237e0   2.37E2   0.000237E+6   2370E-1   0.0000000237E10
```

In such floating-point format, the position of the decimal point varies (floats) depending on the integer following the E.

2.3 RECORD Data Types

The record is a convenient way to organize several related data items of different types. A record consists of one or more fields, each containing one or more data items. The fields of a record can be of different types. The record syntax specifies the name and type of each field.

Syntax

```
RECORD
field-list
END
```

The syntax of a field-list is:

$$\left\{ \begin{array}{l} \{\{\text{field-identifier}\} ,\ldots : [\![\{\text{attribute}\} ,\ldots]\!] \text{ type}\} ;\ldots \\ [\![;\text{variant-clause}]\!] \ [\![;]\!] \\ \text{variant-clause } [\![;]\!] \end{array} \right\}$$

field-identifier
The names of one or more fields.

attribute
BIT, BYTE, POS, READONLY, UNSAFE, VOLATILE, WORD, and WRITEONLY attributes provide additional information (see Chapter 10).

type
The type identifier or type definition of the corresponding field(s). A field can be any type.

variant-clause
The variant part of the record (see Section 2.3.1).

The POS attribute, which can be applied only to a field of a packed record, allows you to position record fields relative to the beginning of the record (see Chapter 10).

The identifiers of the fields must be unique within the record but can be repeated in different record types or subrecords of the same record. For instance, you may define the field Name only once within a particular record type. Other record types, however, could also have fields called Name. The scope of field identifiers within a record type is the record type definition itself.

The values for the fields are stored in the order in which the fields are defined.

Examples

1. The values for the fields of the variable record Team_Rec would be stored in the order Wins, Losses, Percent.

```
VAR
  Team_Rec:RECORD
             Wins : INTEGER;
             Losses : INTEGER;
             Percent : REAL;
           END;
```

2. You refer to a field within a record by specifying the identifier of the record variable and the identifier of the field, separated by a period. For example, the three fields of the record Team_Rec are Team_Rec.Wins, Team_Rec.Losses, and Team_Rec.Percent. You can specify a field anywhere in the program that a variable of the field type is allowed. Thus, you could write:

```
Team_Rec.Wins := 9;

Team_Rec.Losses := 4;
```

3. Records can include fields that are themselves records.

```
VAR
  Order:RECORD
          Part : INTEGER;
          Received : RECORD
                       Month : (Jan, Feb, Mar, Apr, May,
                       Jun, Jul, Aug, Sep, Oct, Nov, Dec);
                       Day : 1..31;
                       Year : INTEGER;
                     END;
          Inventory : INTEGER;
        END;
```

The fields in record Order are Order.Part, Order.Received.Month, Order.Received.Day, Order.Received.Year, and Order.Inventory.

2.3.1 Variant Clause

Variants are fields or groups of fields that can contain different types or amounts of data. Thus, two variables of the same record type can contain different types of data. To specify a variant, include a variant clause in the record type syntax. The variant clause must be the last field in the record.

Syntax

CASE ⟦tag-field :⟧ tag-type OF { case-label-list : (field-list) } ;...

tag-field

The current variant of the record. This field in the record is common to all variants. You can reference the tag-field in the same way that you use any other field in the record. The tag-field may not have attributes associated with it.

tag-type

The ordinal type of the variant field.

case-label-list

One or more constants of the tag-type.

field-list

The names of one or more fields (see Section 2.3).

Examples

1. Assume that the types Name and Day have already been defined. In this example, the last three fields in the record type vary, depending on whether the part is on order. The tag-field identifier Onorder is defined in the variant clause. Records for which the value of Onorder is TRUE will contain information about the current order. Those for which the value of Onorder is FALSE will contain information about the previous shipment.

```
VAR
  Order:RECORD
          Part : 1..9999;
          Stock_Quantity : INTEGER;
          Supplier : Name;
          CASE Onorder : BOOLEAN OF
            TRUE :
              (Promised : Day;
              Order_Quantity : INTEGER;
              Price: REAL);
            FALSE :
              (Last_Shipment : Day;
              Rec_Quantity : INTEGER;
              Cost : REAL);
        END;
```

2. The second way of specifying the tag field is to use only a tag type, as in this example. (Assume that Sex is an enumerated type with values Male and Female.) Here you must keep track of the currently valid variant.

```
VAR
  Persons:RECORD
            Patient : Name;
            Birthdate : Date;
            Age : INTEGER;
            CASE Sex OF
              Female : (Births : 1..30);
              Male : ();
          END;
```

You can define a variant only for the last field in the record. Variant fields, however, can be nested.

Example

```
VAR
  Patients:RECORD
            Patient : Name;
            Birthdate : Date;
            Age : INTEGER;
            CASE Parsex : Sex OF
              Male : ();
              Female : (CASE Births : BOOLEAN OF
                          FALSE : ();
                          TRUE : (Nofkids : INTEGER));
          END;
```

A variable of this record type can contain a patient's name, birth date, age, and sex. In addition, the variable includes a variant field for each woman, based on whether she has had any children. A second variant, containing the number of children, is defined for women who have given birth.

Implementation Notes

- Neither the compiler nor the Pascal OTS verifies that the use of a particular variant, through one of its fields, is consistent with the value of the tag field. Though it is possible to access the fields of different variants without regard to the tag-field value, it is not recommended.
- The compiler does not verify that the case-label value contained in the tag field is the same as the case label associated with the current field references.
- A program must not reference one variant when the record contains the data of another variant. Programs that violate this restriction to facilitate the reinterpretation of a data item must never reference an unpacked field of a variant in an expression or statement containing a packed operand (field or array element); invalid data may result.

2.3.2 Record Examples

1. This example shows a record with six fields.

```
TYPE
  Tax = RECORD
          Year : INTEGER;
          Gross : REAL;
          Net : REAL;
          Deductions : INTEGER;
          Itemized : BOOLEAN;
          Interest : ARRAY[1..5] OF REAL;
        END;
```

   To write a structured constant (Section 3.2.2) for a record of this type, specify its type and include in parentheses a value of the appropriate type for each field. Because the array Interest is nested inside the record, you must include a structured constant for the array within this structured constant, as the following example shows (see Section 2.4 for details):

```
Tax (1979, 10000.0, 8000.0, 1500, FALSE, (5 OF 0.05))
```

2. This example shows a record nested within another record.

```
TYPE
  Name_Data = RECORD
                Name : PACKED ARRAY[1..20] OF CHAR;
                Address : RECORD
                            Number : INTEGER;
                            Street, Town : PACKED
                            ARRAY[1..20] OF CHAR;
                          END;
                Age : 0..150;
              END;
```

2.4 ARRAY Types

An array is a group of components of the same type. You refer to each component of the array by the array identifier and an index. The array syntax specifies an index type and a component type.

Syntax

Form 1:

ARRAY [{ index-type } ,...] OF component-type

Form 2:

ARRAY

[index-type] { OF ARRAY [index-type] OF } ,... component-type

index-type

The type of the index; the type can be any ordinal type except LONG_INTEGER.

component-type

The type of the array components.

The components of an array can be of any type. For example, you can define an array of integers, an array of records, or an array of real numbers. An array of arrays is a multidimensional array, as described in Section 2.4.1.

The indexes of an array must be of an ordinal type. You cannot specify the type INTEGER as the index type, because such an array would exceed the available memory space. To use integer values as indexes, you must specify an integer subrange.

The range of the index type establishes the size of the array and the way in which it is stored. Suppose that the variable Let_1 is an array with 10 components of type CHAR with index 1..10. You refer to the components as Let_1[1], Let_1[2], Let_1[3], and so on through Let_1[10].

You can use array components in any expressions in which you can use variables of the component type. An array can be passed as a parameter and can be the result type of a function. The only operation defined for the array as a whole is the assignment (:=) operation. An exception to this rule is character strings, as described in Section 2.4.2.

2.4.1 Multidimensional Arrays

An array with more than one index is multidimensional; its components are of an array type. An array can have any number of dimensions, and each dimension can have a different index type.

Examples

1. The following is a definition of a 2-dimensional array:

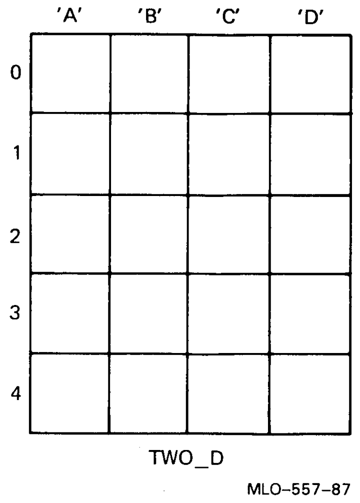
```
TYPE
  Two_D=ARRAY [0..4] OF ARRAY ['A'..'D'] OF INTEGER
```

2. You can abbreviate that syntax by specifying all the index types in one pair of brackets:

```
TYPE
  Two_D=ARRAY [0..4,'A'..'D'] OF INTEGER
```

To refer to a component of the array Two_D, you specify two indexes (one integer and one character) in the order they were declared: Two_D[0,'A'], Two_D[0,'B'], and so on. You can also use the alternative form Two_D[0]['A']. The first index indicates the rows of the array; the second index indicates the columns. Figure 2–1 represents the array Two_D.

Figure 2–1: A 2-Dimensional Array



When you refer to the components of Two_D, the first component in the first row is Two_D[0,'A']. The second component in this row is Two_D[0,'B']. The first component in the second row is Two_D[1,'A']. The last component in the last row is Two_D[4,'D']. In general, element j of row i is Two_D[i,j].

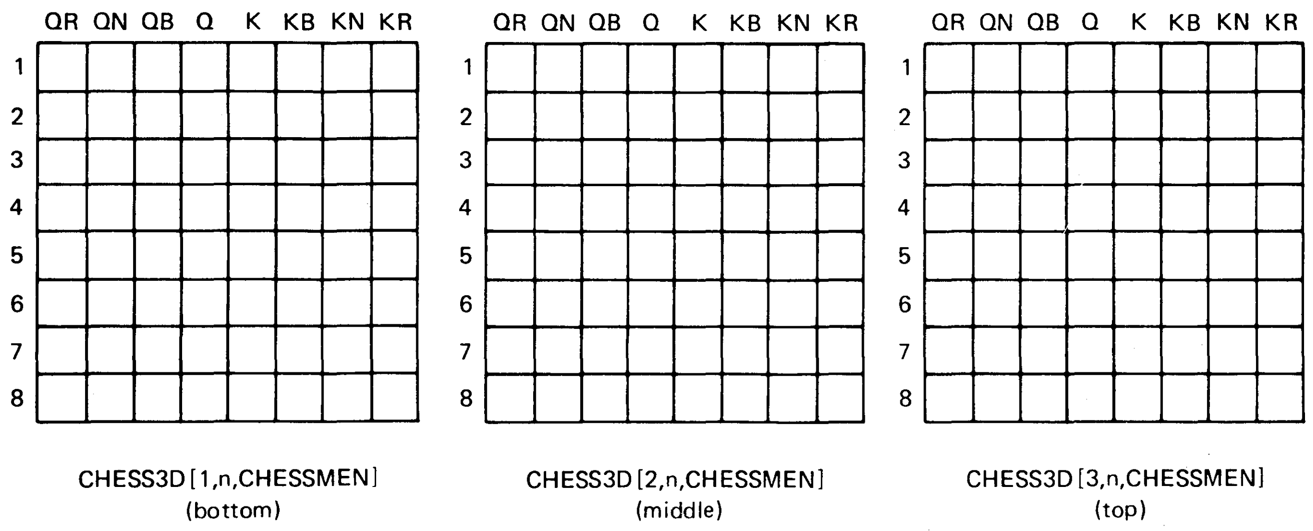
You construct arrays of three or more dimensions in a similar fashion.

Example

The array below specifies a 3-dimensional chessboard whose indexes are the levels, ranks, and files of the chessboard.

```
TYPE
  Chessmen=(QR,QN,QB,Q,K,KB,KN,KR,P,E);
VAR
  Chess3d : ARRAY[1..3, 1..8, QR..KR] OF Chessmen;
```

The reference Chess3d[1] indicates one level or a single chessboard. The reference Chess3d[1,1,QR] specifies the first level, first square in the upper left corner (bottom level, first rank, Queen's Rook file). Figure 2–2 illustrates the three levels of this array.



MLO–558–87

2.4.2 String Types

A character string type in Pascal is a single-dimensional packed array of characters with a lower bound of 1. The length of the string is always fixed and is established by the upper bound of the array's index type. A string constant (see Section 3.2.2.1) is a special case of an implicitly defined packed array of characters.

Example

You could associate the identifier Name with an array of the following description:

```
VAR
  Name : PACKED ARRAY [1..20] OF CHAR;
```

That association would allow you to store a string of 20 characters in the array variable Name. The length of the string must be exactly 20 characters. Pascal neither adds blanks to extend a shorter string nor truncates a longer string. If you specify a string of incorrect length, an error occurs.

2.4.3 Array Examples

1. This example shows a 50-component array of integers in the subrange from 0 to 200.

   ```
   TYPE
     Abc = ARRAY[1..50] OF 0..200;
   ```

   If you wanted to write a structured constant (see Section 3.2.2) to give all the components the value 0, you would write Abc(50 OF 0).

2. This example shows a 2-dimensional array representing a chessboard.

   ```
   TYPE
     Board = ARRAY[1..8,QR..KR] OF Chessmen;
   ```

Assume that the type of the array, Chessmen, is an enumerated type with values QR, QN, QB, Q, K, KB, KN, KR, P, E. You could write the following structured constant to show how the chess pieces are arranged on the board at the start of a game:

```
Board((QR,QN,QB,Q,K,KB,KN,KR), (8 OF P), 4 OF (8 OF E),
(8 OF P), (QR,QN,QB,Q,K,KB,KN,KR))
```

The pieces from Queen's Rook (QR) to King's Rook (KR) are lined up along each end of the board, in the first and eighth rows of the array. The second and seventh rows of the array contain Pawns (P). The third through sixth rows are empty (E).

3. Suppose that you have an array of this description:

```
TYPE
  String=PACKED ARRAY[1..10] OF CHAR;
```

   You could write the following string constants:

```
'C.P.E.Bach'
'engrossing'
Beta = (10 OF ' ')
```

2.5 SET Types

A set is a collection of data items of the same ordinal type (called the base type). The set type definition specifies the values that can be members of sets of that type.

Syntax

SET OF base-type

base-type
The identifier or definition of the type from which the members of sets of this type are selected. Any ordinal type except LONG_INTEGER may be used.

You define a set by listing all the values that can be its members. A set can have a maximum of 256 elements. In the case of INTEGER and UNSIGNED values, the value of each element must be between 0 and 255. Values outside that range therefore cannot be set members. For sets of other ordinal base types, members can include the full range of the type, provided no more than 256 members are in the range.

You initialize a set variable by using a set constructor (see Section 3.4) in an assignment statement. The constructor specifies one or more values of the base type.

Examples

1. This example shows a set with individual characters as members.

```
TYPE
  Alphabet = SET OF CHAR;
```

   You could write the following constructors for a set of this type:

```
['A','E','I','O','U']
['B'..'D','F'..'H','J'..'N','P'..'T','V'..'Z']
```

2. This example shows a set with an integer subrange as its base type.

```
TYPE
  MAP = SET OF 1..255;
```

A constructor for this set might include the following values:

```
[3,4,15,20,23,34,40,45,55,60,70]
```

The upper limit of this subrange is the maximum allowed for sets.

2.6 FILE Types

A file is a sequence of components of the same type. The number of components in a file is not fixed; a file can be of any length.

Data of a file type is intended for output to or input from an environment that is external to a program. Typical external file environments are terminals, disks, digital-to-analog converters, and other programs residing on the same or remote processors.

The file type definition identifies the component type.

Syntax

FILE OF component-type

component-type

The type of the file's components. Any scalar or structured type may be used except a file type or a structured type containing a component that is of a file type.

A variable declared to be of a file type is called a file variable and identifies the file. You may not use a file variable or a structure containing file components in an expression or in an assignment statement. Therefore, you cannot assign one file variable to another. Likewise, you cannot compare two arrays that have file components or form structured constants of a file type.

Type compatibility for file variables applies only to those that are passed to a routine. You can pass a file only as a VAR parameter (see Section 6.6.4).

When you declare a variable of a file type, Pascal automatically declares a buffer variable of the component type. That variable takes on the value of one file component at a time. The predeclared I/O requests, described in Chapter 9, move the file position, thus changing the value of the buffer variable. That buffer variable is given a name composed of the file variable's identifier suffixed by either the circumflex (^) or the at (@) symbol. For example, if Math_Scores is a FILE OF INTEGER, Pascal creates Math_Scores^ as an integer buffer variable associated with the file Math_Scores.

Figure 2–3 illustrates the buffer variable contents after execution of a RESET I/O request and again after a READ I/O request.

Figure 2–3: Buffer Variable Contents After Using READ and RESET

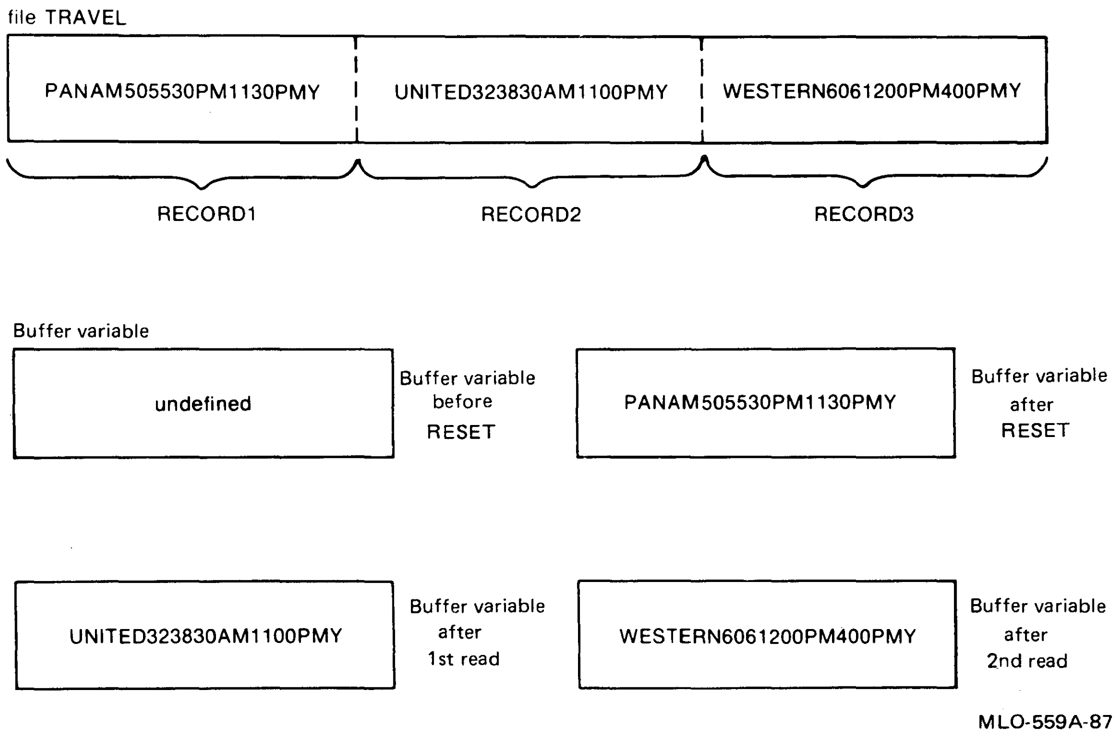
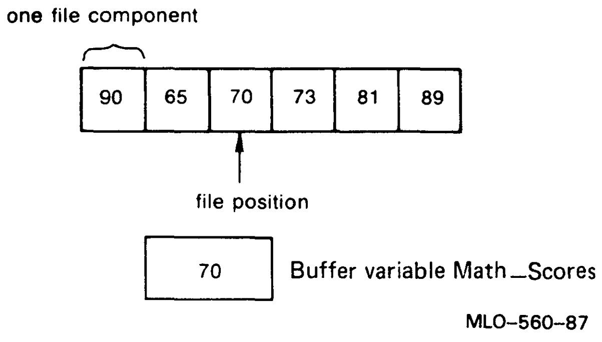


Figure 2–4 illustrates the contents of the buffer variable during the use of the file Math_Scores.

Figure 2–4: Buffer Variable Contents During Use



Examples

1. This example shows a file of Boolean values.

   ```
   VAR
     Truthvals : FILE OF BOOLEAN;
   ```

   If you give this file the variable identifier Truthvals, you would refer to the buffer variable as Truthvals^.

2. The components of the following file are character strings 20 characters in length:

   ```
   TYPE
     Alpha20 = FILE OF PACKED ARRAY[1..20] OF CHAR;
   ```

   You could create variables of this file type to contain lists of names, such as Accept_List, Reject_List, and Wait_List.

3. This example shows a file of records.

   ```
   TYPE
     Sample = FILE OF
              RECORD
                Trial : INTEGER;
                Date : RECORD
                         Month : (Jan,Feb,Mar,Apr,May,Jun,Jul,Aug,Sep,Oct,Nov,Dec);
                         Day : 1..31;
                         Year : INTEGER;
                       END;
                Temp, Pressure : INTEGER;
                Yield, Purity : REAL;
              END;
   ```

   If this file had the variable identifier Results, you would access fields of the record components as Results^.Trial, Results^.Date.Month, and so on.

2.7 TEXT File Type

Pascal supplies a predefined file type called TEXT. Variables of file type TEXT are called text files and consist of components of type CHAR.

Unlike a FILE OF CHAR, the components of a file of type TEXT are grouped into sequences or lines of characters terminated by an end-of-line marker. You can refer to the marker indirectly through the READLN and WRITELN procedures and the EOLN function (see Sections 9.19 and 9.27).

Note

The NULL ASCII code (CHR(0)) is not valid in a TEXT file and is ignored on input.

The predeclared file variables INPUT and OUTPUT are files of type TEXT. The variables refer to the standard Pascal input and output files, which ordinarily communicate with a terminal. Those files are the defaults for all the predeclared TEXT file procedures (see Chapter 9).

2.8 Pointer Types

Variables normally exist for the lifetime of the program or the routine in which they are declared. Program-level variables are allocated in static storage, whereas routine-level variables are allocated automatically on the stack. Some applications, however, require variables that have shorter lifetimes or an unknown number of variables of a certain type. Pascal allows you to declare variables of the pointer type, called dynamic variables, to fill those requirements.

During program execution, dynamic variables are allocated from an area called heap storage as they are needed. Unlike other variables, dynamic variables do not have identifiers; you must refer to them indirectly with pointers. A variable of the pointer type may have as its value either the address of a dynamic variable or the predefined identifier NIL.

Syntax

$\left\{ \begin{matrix} \hat{} \\ @ \end{matrix} \right\}$ base-type-identifier

base-type-identifier

The type of the dynamic variable to which the pointer type refers. The base type can be any type except a file type.

A pointer assumes a value by:

- Assigning the pointer the value of another pointer of the same type
- Assigning the value of the NIL constant
- Using the ADDRESS function 8.3
- Using the GET_PACKET procedure 14.15
- Using the NEW procedure 8.14

The constant NIL indicates that the pointer does not specify an address. Thus, a NIL pointer does not point to a variable.

Note

At execution time, the system can verify that a pointer does not have a NIL value but the system cannot check for other illegal addresses.

The NEW and DISPOSE procedures, described in Sections 8.14 and 8.8, allocate and deallocate dynamic variables.

Variables of a pointer type point to variables of the base type and are said to be bound to that type. To indicate a pointer variable, specify its name. To indicate the dynamic variable to which a pointer is bound, specify the pointer name, followed by a circumflex (^) or an at (@) sign. For example, suppose that M is a pointer variable bound to records of type Myrec. You specify M^ to denote the record variable to which M points.

In the only exception to the rule for declaring an identifier, Pascal allows you to use the base type identifier in a pointer type definition before you define the base type. However, the base type must be defined before the end of the TYPE section in which it is first mentioned.

Example

This example defines a pointer to a user-created type Movie.

```
TYPE
  Ptr_to_Movie = ^Movie;
```

Later in the same TYPE section, you could define a record type with a component of type Ptr_to_Movie.

The MicroPower/Pascal language allows you to define pointers to types containing files. The files referenced by a pointer are not closed until execution of the program terminates. If you do not want the files to remain open throughout program execution, you must use the CLOSE procedure (see Section 9.4).

Example

Suppose that the following record type is associated with the type identifier Hits and that another type identifier, Ptr_to_Hits, is declared to be of type ^Hits:

```
VAR
  Rec_Info:RECORD
             Title, Artist, Composer : ARRAY [1..30] OF CHAR;
             Weeks_On_Chart, NSold : INTEGER;
             First_Version : BOOLEAN;
           END;
```

You could then create an array variable Topten of the type:

```
ARRAY [1..10] OF Ptr_to_Hits;
```

The variable Topten would thus have 10 components of a pointer type to refer to the record type Hits. You could use this array to create a table of pointers to 10 records of type Hits.

2.9 Type Checking Rules

Pascal enforces type checking for identical types and compatible types. Identical-type rules determine the types of data you can pass as VAR parameters and the types of pointer assignments you can make. Compatible-type rules determine the types of values you can assign to variables of each type or pass as value parameters.

2.9.1 Identical Types

Types are identical if they have the same type identifier or an alias thereof. Pascal requires that a variable passed as an actual parameter to a routine be identical to the corresponding formal VAR parameter. Pascal also checks that the base types are identical when you assign a pointer expression to a pointer variable.

Type A is identical to type B if one of the following conditions is true:

- A and B have the same type identifier. For example, type C is defined first or is predefined, and A and B are both of type C.
- A and B have been made equivalent by a type definition. For example, type A is defined first or is predefined; then the definition B = A is made.

- A and B have been made equivalent to two type identifiers that were previously made equivalent in a type definition. For example, type C is defined first or is predefined; the definition D = C is then made; finally, the definitions A = D and B = C are made.

2.9.2 Compatible Types

Compatible-type rules apply to the values with which you initialize variables, values assigned using the assignment statement, and actual parameters passed to formal value parameters. The contexts in which an expression is assignment compatible with a variable of the same or a different type are listed below.

Source Type	Destination Type
INTEGER	INTEGER or integer subrange, LONG_INTEGER, UNSIGNED or unsigned subrange, REAL type
LONG_INTEGER	LONG_INTEGER or REAL
UNSIGNED	UNSIGNED or unsigned subrange, LONG_INTEGER, REAL type
CHAR	CHAR
Subrange	Base type
REAL	REAL type
PACKED ARRAY OF CHAR	PACKED ARRAY OF CHAR
Pointer	Pointer to identical type

Rules

- Two record types are compatible only if they are identical (see Section 2.9.1).
- Two array types are compatible only if they are identical or if they are single-dimensional PACKED arrays of CHAR with the same number of elements.
- A set expression is compatible with a set variable if the base types are equivalent. In addition, all members of the expression must be included in the range of the variable's base type.
- Assignment operations are not allowed on file types and on structured types having file components.

Certain attributes affect the rules of compatible types. (See Chapter 10 for a complete discussion of attributes.) The resulting changes to compatible-type rules are as follows:

- READONLY—No expression is compatible with either a READONLY variable of equivalent type or a variable with a READONLY component of equivalent type.

- UNSAFE—An expression of any type is compatible with an UNSAFE variable unless the variable has the READONLY attribute. If the expression type and the variable type have the same allocation sizes, Pascal assigns the value of the expression with no type conversion. If the representation of the expression is larger than the UNSAFE variable, the compiler generates a diagnostic error. If the representation of the expression is smaller than the UNSAFE variable, the compiler leaves the high-order bits of the UNSAFE variable unchanged.

Chapter 3
Expressions

An expression is a range of value descriptions. An expression is an operand or a combination of operands and operators. Operands include constants, variables, function identifiers, set constructors, and subexpressions, which are expressions enclosed in parentheses. Examples of operands are:

Operand	**Type**
123.34	Numeric constant
'x'	Character constant
TRUE	Boolean constant
Average (3.3, 'T', 9)	Structured constant
'TEST.TXT'	String constant
numberofpeople	Identifier of a variable or a constant
students.grade	Identifier of a record field
master[a,b]	Identifier of a subscripted variable
(xc+bd)	Subexpression
SQRT(newval)	Function identifier
[i, elt2]	Set constructor

The operators in an expression represent predefined arithmetic, relational, logical, string, and set operations.

Examples

```
AZ / VST      Contents of AZ divided by the contents of VST
A = B         Contents of A compared with the contents of B
```

Pascal recognizes compile-time expressions and run-time expressions. A compile-time expression consists of one or more elements that can be evaluated when the program is compiled. The simplest compile-time expression is a single constant or a constant identifier. Other compile-time expressions combine constants and constant identifiers with Pascal operators and certain predeclared functions (see Section 4.1).

A run-time expression consists of at least one element that cannot be evaluated until the program is executing. Run-time expressions contain variables and function identifiers and can also include constants, constant identifiers, Pascal operators, and predeclared functions.

You can use arithmetic, relational, logical, string, and set operators to form Pascal expressions. Those operators are explained in Sections 3.5.1 to 3.5.5. The order in which Pascal evaluates the components of an expression is determined by the precedence rules for the operators and operands, as outlined in Sections 3.6 and 3.7.

When forming an expression in Pascal, you are not limited to combining integers only with integers, real numbers only with real numbers, and so forth. Pascal promotes data from one type to another under certain circumstances, presented in Section 3.8, so you can form expressions with data of different types.

Although you cannot change the type of a variable once it has been declared, you might want to have this capability when forming expressions. The MicroPower/Pascal language allows you to alter temporarily your concept of a variable's type by using the type cast operator, explained in Section 3.9.

3.1 Variables

A variable is a symbolic representation of a data storage location. Variables contain data that may be altered during a program's operation. You declare the name of each variable that you use and associate it with a particular data type.

Variables may contain either scalar data or structured data (see Chapter 2). The data in a variable must conform to the requirements of the data type definition with which the variable is declared. The declaration may be either in the VAR part of the declaration section of a program, module, or subprogram (see Chapter 4) or in the formal parameter list of a subprogram (see Chapter 6).

3.2 Constants

A constant is the explicit or symbolic representation of a value that does not change during the execution of a program. The two kinds of constants in the MicroPower/Pascal language are scalar constants and structured constants.

You may refer to a constant in two ways. You may write it explicitly by specifying the value itself. Or, you may declare a constant in a CONST declaration so you can refer to it symbolically by an identifier (called a constant identifier).

3.2.1 Scalar Constants

Scalar constants are constant values of the scalar data types (INTEGER, LONG_INTEGER, UNSIGNED, REAL, CHAR, BOOLEAN) and user-declared enumerated types. The ranges of numeric constant values and their associated data types are:

Constant in Range	Resulting Type
-2147483647 .. -32769	LONG_INTEGER
-32768 .. 32767	INTEGER
32768 .. 65535	UNSIGNED
65536 .. 2147483647	LONG_INTEGER
$-2.938736 \times 10^{-39}$ .. -1.701411×10^{38}	REAL
2.938736×10^{-39} .. 1.701411×10^{38}	REAL

Example

In the following example, 12, 64, 'T', 3.4, A, B, 2, and Constchar are scalar constants:

```
CONST
A = 12;
B = 64;
Constchar = 'T';
    .
    .
    .
12+3.4

A DIV 2

Constchar
```

3.2.2 Structured Constants

Structured constants are constant values of the RECORD and ARRAY types. You specify a structured constant by using the following syntax:

Syntax

$$\left\{ \begin{array}{l} \text{type-identifier (\{ value \} ,...)} \\ \text{string-constant} \end{array} \right\}$$

type-identifier

The identifier of the data type of the structure to which this constant applies. If the type-identifier describes a packed array of type CHAR, you must explicitly specify the values for each element of the array as character constants.

value

A constant, the sequence "integer-constant OF value", or a list of the form "(" value {,value} ... ")". The form of this parameter is determined by the structure specified by type-identifier.

The value(s) specified must be consistent with the components of the structure specified by type-identifier.

string-constant

A character string. This is a special case of structured constant because a string has an implied type of PACKED ARRAY [1..n] OF CHAR (n is the length of the string). Use this form of structured constant for string types (see Sections 2.4.2 and 3.2.2.1).

Rules and Defaults

- You must specify a value for every component in the structure. For records, the component values must also be of the same type as the fields in which they reside. For arrays or records with multiple fields, the constants must be specified in the same order in which they were declared.
- You may not specify a set constructor (see Section 3.4) as a value in a structured constant. When a variable of a structured data type includes a set, its components may receive values only by explicit assignment.
- You can use the OF repetition factor to specify the same constant for consecutive components.
- Structured constants for nested records or nested arrays must be nested to the same level as specified by the record or array declaration and be delimited by parentheses. If the nested record has a type identifier, that identifier cannot appear in the constant.
- A structured constant for a variant record must include constants for the tag fields that specify the variants.
- A value in a structured constant will never have its type converted to conform to the type of a structured component. Conversion rules for operands in expressions do not apply. For example, you cannot use an INTEGER value where a REAL value is expected. You can, however, use a value of less than 65,536 where a LONG_INTEGER value is expected, because integers below that value are promoted to LONG_INTEGER values, if required.

Examples

1. This example shows how you can use string types with structured constants.

```
TYPE
  String = PACKED ARRAY[1..10] OF CHAR;

CONST
  Name = 'JEFFERSON ';

VAR
  String_var : String;
```

You can write assignment statements such as:

```
String_var := Name;
String_var := String ('J','E','F','F','E','R','S','O','N',' ');
String_var := 'JEFFERSON ';
```

2. This example shows a record nested within another record.

```
TYPE
  Voter = RECORD
            Party : (Democrat,Republican,Independent,Other);
            Registered : BOOLEAN;
            Name_Data : RECORD
                          Name,Street,Town : PACKED ARRAY[1..16] OF CHAR;
                          Zip : PACKED ARRAY[1..5] OF CHAR;
                        END;
          END;

VAR
  P : Voter;
```

   Given this record type, you could write the following structured constant:

```
P := Voter (Democrat,TRUE,('       Adam Smith',
                           '114 Birdhouse Rd',
                           '    Bird-in-Hand',
                           '06312'));
```

   Note the use of parentheses to specify record fields that are nested records.

3. Suppose that you have the following record type:

```
TYPE
  Calltyp = RECORD
              Caller : PACKED ARRAY[1..10] OF CHAR;
              Time : REAL;
              Subj : (Work, Play, Sales, Chat);
              CASE BOOLEAN OF
                TRUE : (HOUR : INTEGER);
                FALSE : ();
            END;

VAR
  Call : Calltyp;
```

   You could initialize a variable of this record type by using the following structured constant:

```
Call := Calltyp ('Washington', 10.30, Chat, TRUE, 12)
```

   The constant provides a string constant for the field Caller, a real number for the field Time, and the constant identifier Chat for the field Subj. The structured constant specifies the tag field with the Boolean value TRUE and the variant field Hour with the integer value 12. Note that the tag field is specified even though it does not have an identifier.

   To specify a constant for this record with the value FALSE for the variant's tag field, you could write the following structured constant:

```
Call := Calltyp ('Washington', 10.30, Chat, FALSE)
```

   This constant specifies the same values as the previous one for all fields except the tag field. The tag field value is now last in the list because the FALSE case of the variant specifies no additional fields.

4. Suppose that you have the following array:

```
TYPE
   Average = ARRAY[1..10] OF REAL;
```

You could specify the following structured constant:

```
Average (1.318, 4.2029, 2 OF 3.68, 2 OF 9.6445, 7.0, 3 OF 5.772)
```

The structured constant specifies the value 1.318 for the first component of the array, 4.2029 for the second component, and 7.0 for the seventh component. The structured constant includes the repetition factor 2 OF 3.68, which specifies the value 3.68 for the third and fourth components and the repetition factor 2 OF 9.6445, which specifies the value 9.6445 for the fifth and sixth components. The repetition factor 3 OF 5.772 specifies the value 5.772 for the last three components.

5. This example shows a structured constant for the array Students of component type Student_Range.

```
TYPE
  Student_Range = 1..50;
  Students = ARRAY[1..10] OF Student_Range;
```

The type identifier is optional when it is nested inside another structured constant.

```
Students(25,22,23,20,35,17,29,31,20,26)
```

6. This example shows a 2-dimensional array of real numbers:

```
TYPE
  Twoby = ARRAY[0..3] OF ARRAY[1..5] OF REAL;
```

A structured constant for that array must consist of four structured constants, each having five real values.

```
Twoby ((1.0,1.1,1.2,1.3,1.4), 2 OF (5 OF 0.0),(10.1, 2 OF 11.0, 2 OF 11.1))
```

If you visualize the first index of this array as representing rows and the second index as representing columns, the structured constant above is filling the columns of the array one row at a time. Figure 3–1 illustrates the assignment of those constants to an array variable.

Figure 3–1: Values Assigned to a 2-Dimensional Array

	1	2	3	4	5
0	1.0	1.1	1.2	1.3	1.4
1	0.0	0.0	0.0	0.0	0.0
2	0.0	0.0	0.0	0.0	0.0
3	10.1	11.0	11.0	11.1	11.1

MLO–559–87

7. You initialize arrays of three or more dimensions with structured constants, using the same syntax as shown in the previous example.

```
TYPE
  Letters = ARRAY[1..2,1..3,1..4] OF CHAR;
```

Given the array above, you could write the following structured constant:

```
Letters (2 OF (3 OF(4 OF'*')))
```

8. Suppose you have the following record:

```
TYPE
  Datarec = RECORD
              Kids : INTEGER;
              Deviation : REAL;
              Code : CHAR;
              Verified : BOOLEAN;
              Name : (Washington, Lincoln, Jefferson, Adams);
            END;
```

   The following example shows a structured constant for a variable of this record type:

```
Datarec (17, 3.2075, 'P', FALSE, Adams)
```

9. You can declare a constant as a structured constant and reference its components as you would a structured variable.

```
TYPE cities = (NY,SF,LA,BOS,CHI,CLE,STL,KAN);
     cities_array = ARRAY[1..8] OF cities;

CONST tax_rank = cities_array(NY,BOS,SF,LA,CHI,STL,CLE,KAN);
      .
      .
      number_one := tax_rank[1];
      .
      .
```

3.2.2.1 String Constants

A string constant is a sequence of characters enclosed in apostrophes (′ ′). A string constant is a special case of structured constant. The type of a string constant is implicitly defined as a packed array of characters that is the length of the specified string.

Syntax

′ ⟦{ character } ...⟧ ′ ⟦⟦ ({ integer } ,...)⟧ ⟦ string-constant ⟧⟧

character
A graphic ASCII character, a space character, or a tab character.

integer
An integer constant in the range 0 to 255 that specifies an ASCII character code. The integer must not be a constant identifier. This syntax allows you to specify any ASCII character, including the nonprinting characters, as a string constant.

string-constant
A string constant.

You must specify at least one character or integer. Null strings are not allowed.

Examples

1. The following string constants are implicitly defined as a packed array with length 20. If the string has fewer than 20 characters, you must add characters to extend the string.

   ```
   'Indianapolis,Indiana'
   'Milwaukee, Wisconsin'
   'Paris, France       '
   ```

2. The ordinal value of the bell character is 7, whereas the value of the null character is 0. The integers 7 and 0 are enclosed in parentheses within the character string.

   ```
   'A bell '(7)' in a null-terminated ASCII string'(0)
   ```

3. In this example, CR illustrates an illegal use of a constant identifier.

   ```
   CONST
     CR = %O'15';
        .
        .
        .
   WRITELN ('INVALID'(CR)'USE');
   ```

3.3 Function Identifiers

A function identifier is the name of a group of statements declared to be a function (see Chapter 6). The function identifier is an expression operand that represents a value derived from the execution of the statements that it identifies.

3.4 Set Constructors

A set constructor is an expression that specifies the members of a particular set. The set constructor consists of a list of scalar elements of the same base type or an empty list.

Syntax

[{ element } ,...]

element

A variable, constant, or expression that specifies a value of the base type. You can specify constants that appear consecutively in the set definition by using the subrange (..) symbol. If no element is specified, the set is empty.

You use an assignment statement (see Section 5.1) to assign the values specified by a set constructor to a set variable. The base type of the variable must include all members of the set to which the expression evaluates. You can use a set constructor with the set operators (see Section 3.5.5) to form expressions.

A set having no elements is called an empty set and is written [ ].

Examples

1. This example shows a set with individual characters as members.

   ```
   TYPE
     Alphabet=SET OF CHAR;
   VAR
     Vowels, Consonants: Alphabet;
   BEGIN
     Vowels := ['A','E','I','O','U'];
     Consonants := ['B'..'D','F'..'H','J'..'N','P'..'T','V'..'Z'];
   END.
   ```

   The set variables Vowels and Consonants are of the same base type but are initialized to contain the appropriate characters as specified in the set constructors.

2. This example shows a set with an integer subrange as its base type.

   ```
   TYPE
     MAP=SET OF 0..255;
   ```

   A constructor for this set might include the following values:

   ```
   [3,4,15,20,23,34,40,45,55,60,70]
   ```

   The upper limit of this set's subrange is the maximum number of elements allowed for sets.

3. This example shows a constructor for a set of integers ranging from 35 to 115.

   ```
   [39, 67, 95, 110..115]
   ```

   The constructor specifies nine constants: 39, 67, 95, and all the integers between 110 and 115.

3.5 Operators

Operators let you form complex compile-time and run-time expressions to combine constants, constant identifiers, variables, and function identifiers. Operators can be either dyadic or monadic. A dyadic operator has two operands, as demonstrated by the + operator in the expression A+B. Most Pascal operators are dyadic operators. A monadic operator has one operand, as demonstrated by the "–" operator in the expression –B. The three monadic operators are identity (+), sign inversion (–), and NOT.

The Pascal operators are classified as follows:

- Arithmetic
- Relational
- Boolean
- String
- Set

3.5.1 Arithmetic Operators

An arithmetic operation usually provides a formula for calculating a value. To construct an arithmetic expression, you combine numeric constants, variables, and function identifiers with one or more of the operators from Table 3–1.

Table 3–1: Arithmetic Operators

Operator	Example	Result
+	A+B	Sum of A and B
+	+B	The positive value B
–	A–B	B subtracted from A
–	–B	The negative value B
*	A*B	Product of A and B
/	A/B	A divided by B
DIV	A DIV B	Truncated result of A divided by B
MOD	A MOD B	Modulus of A with respect to B

The addition (+), subtraction (–), and multiplication (*) operators allow operands of the INTEGER, LONG_INTEGER, UNSIGNED, and REAL types. An expression with operands of the same type produces a result of the same type. The result type for an expression with operands of different types is the operand type having the higher precedence (see Section 3.8).

The division (/) operator allows operands of the INTEGER, LONG_INTEGER, UNSIGNED, and REAL types but always produces a REAL result.

The DIV and MOD operators allow operands of the INTEGER, LONG_INTEGER, and UNSIGNED types. DIV divides the first operand value by the second operand value and truncates any fraction from the result. An expression with operands of the same type produces a result of the same type. The result type for an expression with operands of different types is the operand type having the higher precedence (see Section 3.8). For example, the expression 23 DIV 12 equals 1, and –5 DIV 3 equals –1. MOD returns the modulus of the first operand with respect to the second. The operation A MOD B is defined only when B is a positive integer. The result of A MOD B is always an integer from 0 to B–1.

Examples

```
5 MOD 3  (* The result is 2 *)
(-4) MOD 3 = 2  (* The result is 2 *)
2 MOD 5 = 2  (* The result is 2 *)
```

In arithmetic expressions, Pascal allows you to mix INTEGER, LONG_INTEGER, UNSIGNED, and REAL values with subranges of INTEGER and UNSIGNED values. When you assign the value of an expression to a variable, you must make sure that their types are compatible (see Section 2.9.2).

Table 3–2 lists the type of the result for all possible combinations of arithmetic operators and operands.

Table 3–2: Result Types for Arithmetic Operations

First Operand	Second Operand	Multiply (*) Subtract (-) Add (+)	DIV MOD	Divide (/)
INTEGER	INTEGER	INTEGER	INTEGER	REAL
	LONG_INTEGER	LONG_INTEGER	LONG_INTEGER	REAL
	UNSIGNED	UNSIGNED	UNSIGNED	REAL
	REAL	REAL	ERROR	REAL
LONG_INTEGER	INTEGER	LONG_INTEGER	LONG_INTEGER	REAL
	LONG_INTEGER	LONG_INTEGER	LONG_INTEGER	REAL
	UNSIGNED	LONG_INTEGER	LONG_INTEGER	REAL
	REAL	REAL	ERROR	REAL
UNSIGNED	INTEGER	UNSIGNED	UNSIGNED	REAL
	LONG_INTEGER	LONG_INTEGER	LONG_INTEGER	REAL
	UNSIGNED	UNSIGNED	UNSIGNED	REAL
	REAL	REAL	ERROR	REAL
REAL	INTEGER	REAL	ERROR	REAL
	LONG_INTEGER	REAL	ERROR	REAL
	UNSIGNED	REAL	ERROR	REAL
	REAL	REAL	ERROR	REAL

3.5.2 Relational Operators

A relational operation or condition tests the relationship between two arithmetic or Boolean expressions. A relational expression consists of two scalar variables or arithmetic expressions separated by one of the relational operators listed in Table 3–3.

Table 3–3: Relational Operators

Operator	Example	Result
=	A = B	TRUE if A is equal to B
<>	A <> B	TRUE if A is not equal to B
>	A > B	TRUE if A is greater than B
> =	A > = B	TRUE if A is greater than or equal to B
<	A < B	TRUE if A is less than B
<=	A <= B	TRUE if A is less than or equal to B

The two characters in the not-equal (<>), greater-than-or-equal (> =), and less-than-or-equal (<=) operators must appear in the specified order and cannot be separated by a space.

Pascal produces a Boolean result when evaluating a relational expression. Every relational expression therefore evaluates to TRUE or FALSE. For example, the condition 2 <3 is always TRUE; the condition 2> 3 is always FALSE.

3.5.3 Boolean Operators

Boolean operations test the truth value of combinations of conditions. A Boolean expression consists of two or more expressions that have Boolean results, separated by one of the Boolean operators in Table 3–4.

Table 3–4: Boolean Operators

Operator	Example	Result
AND	A AND B	TRUE if both A and B are TRUE
OR	A OR B	TRUE if either A or B is TRUE (or if both are TRUE)
NOT	NOT A	TRUE if A is FALSE (and FALSE if A is TRUE)

The AND and OR operators combine two conditions to form a compound condition. The NOT operator reverses the truth value of a condition, so that if A is TRUE, NOT A is FALSE, and vice versa.

As with relational expressions, the result of a Boolean expression is a Boolean value.

3.5.4 String Operators

You can use the operators in Table 3–5 to compare character string variables and constants. Two character strings are compared character by character in a left-to-right order. The comparison is based on the ASCII collating sequence (see Appendix A).

Table 3–5: String Operators

Operator	Example	Result
=	A=B	TRUE if character strings A and B have equal ASCII values
<>	A <> B	TRUE if character strings A and B have unequal ASCII values
<	A <B	TRUE if ASCII value of character string A is less than that of character string B
<=	A <=B	TRUE if ASCII value of character string A is less than or equal to that of character string B
>	A> B	TRUE if ASCII value of character string A is greater than that of character string B
> =	A> =B	TRUE if ASCII value of character string A is greater than or equal to that of character string B

The string operators are legal only for character strings of the same length. You may not use them to compare two character strings with unequal lengths.

Examples

```
'motherhood' > 'cherry pie'
```

This relational expression is TRUE because lowercase 'm' comes after lowercase 'c' in the ASCII character set. If the first characters in the strings are the same, Pascal looks for differing characters, as in the following:

```
'string1' < 'string2'
```

This expression is also TRUE, because the digit 1 precedes the digit 2 in the ASCII character set.

3.5.5 Set Operators

The set operators let you form sets and test for various set relationships. You can use the operators in Table 3–6 with operands that are set values: set variables, set constructors, and set expressions. The left operand of the IN operator, however, must be a scalar value.

Table 3-6: Set Operators

Operator	Example	Result
+	A+B	Union of sets A and B
*	A*B	Intersection of sets A and B
–	A–B	Set of those elements of set A that are not also in set B
=	A=B	TRUE if set A is equal to set B
<>	A <> B	TRUE if set A is not equal to set B
<=	A <=B	TRUE if set A is a subset of set B
> =	A> =B	TRUE if set B is a subset of set A
IN	A IN B	TRUE if scalar value A is an element of set B

The IN operator requires a set expression as its right operand and a scalar expression of the associated base type as its left operand—for example:

```
2*3 IN [1..10]
```

The value of this expression is TRUE, because 2*3 evaluates to 6, which is a member of the set [1..10].

3.6 Precedence of Operators

The operators in an expression establish the order in which Pascal combines the values. The order of precedence of the operators is listed below, from highest to lowest.

Operator	Precedence
NOT	Highest
*, /, DIV, MOD, AND	↓
unary +, unary –, +, –, OR	↓
=, <>, <, <=, >, >=, IN	Lowest

Pascal evaluates operators of equal precedence (such as + and –) from left to right. You must use parentheses for correct evaluation when you combine relational operators. Consider, for example:

```
A<=X AND B<=Y
```

If no parentheses are used, Pascal attempts to evaluate this expression as A <= (X AND B) <=Y and generates an error. The expression needs parentheses as follows:

```
(A<=X) AND (B<=Y)
```

To evaluate the rewritten expression, Pascal compares the truth values of the two relational expressions.

You can use parentheses in any expression to force a particular order for combining the values. For example:

Expression:	Evaluates to:
8 * 5 DIV 2–4	16
8 * 5 DIV (2–4)	–20

Pascal evaluates the first expression, according to the normal rules for precedence. First, Pascal multiplies 8 by 5 and divides the result (40) by 2. Then Pascal subtracts 4, resulting in 16. The parentheses in the second expression, however, force Pascal to subtract before dividing. Hence, it subtracts 4 from 2, producing –2. Then it divides 40 by –2, with –20 as the result.

Parentheses can also help to clarify an expression. For instance, you could write the first example above as follows:

```
((8 * 5) DIV 2) - 4
```

The parentheses eliminate any confusion about how the expression is to be evaluated.

3.7 Order of Evaluation of Boolean Operands

The order of evaluation of Boolean operands used with the dyadic operators is indeterminate, and the side effects of the evaluations are not predictable. Pascal's ability to produce the correct result is not usually affected, but it is an important consideration when you are performing Boolean operations involving function identifiers that have side effects. (A side effect is an assignment to a nonlocal variable or to a VAR parameter within a function block.) For example, the following IF statement contains two function identifiers for function F:

```
IF F(A) OR F(B) THEN ...
```

Suppose that function F assigns the value of the actual parameter to a nonlocal variable. Because Pascal does not guarantee which function identifier is evaluated first, you cannot be sure of the value of the nonlocal variable after the IF statement is performed.

The compiler uses a "short-circuit" evaluation algorithm on expressions with Boolean operands. Thus, the compiler will not evaluate an entire expression if the compiler obtains an intermediate result that conclusively establishes the final result. Function identifiers, however, are always evaluated.

Example

```
WHILE (I <= Imax) AND (A[I] <> ' ' ) DO I := I+1 ;
```

Since no guarantee is supplied that the evaluation of the compound condition will proceed in left-to-right sequence, the subscripted reference (A[I]) may not always be legal.

3.8 Type Promotion

Since Pascal performs extensive type checking on data, you cannot normally use a value of one type as if the value were of a different type. For example, you cannot assign the character '1' to an integer data item, because '1' is a character value, not an integer value. For the numeric data types, however, the compiler promotes data from one type to another in three instances: arithmetic operations (see Section 3.5.1), assignment statements (see Section 5.1), and when passing actual parameters to formal parameters (see Section 6.6.1). The compiler performs this type conversion according to the following precedence:

```
lowest ————————————————————→ highest

INTEGER < UNSIGNED < LONG_INTEGER < REAL
```

Thus, an INTEGER operand used with an UNSIGNED operand produces an UNSIGNED result. Similarly, a LONG_INTEGER operand used with a REAL operand produces a REAL result.

Note

When converting a negative integer to an unsigned value, the compiler retains the value of the integer's sign bit (bit 15). Because the sign bit is the most significant bit in an unsigned value, negative integers are converted into large unsigned integers.

3.9 Type Cast Operator

The type cast operator (::) lets you change the type characteristics of an expression, variable, or component variable. You may use the type cast operator when you want to mix data objects of different types in contexts where Pascal type-checking rules would prohibit mixing. The type cast operator can follow any expression, variable, or component selector that appears in a statement.

Type casting alters the way data is interpreted for the duration of an operation but does not convert the data of a given variable to a target type. For example, the expression ('1')::INTEGER casts the representation of the string constant '1' as an integer. The resulting value is 49, the decimal value of the ASCII character 1.

Syntax

$$\left\{ \begin{array}{l} \text{(expression) \{ :: type-identifier \}, ...} \\ \text{variable-identifier} \left\{ \begin{array}{l} \text{:: type-identifier} \\ \text{component-selector} \end{array} \right\} , \ldots \end{array} \right\}$$

expression
An expression that is to be type cast.

type-identifier
The identifier of the type to which the object variable or expression is to be cast.

variable-identifier
The identifier of a variable that is to be type cast. The identifier can be an entire variable, an array element, a record field, a buffer variable, a file variable, or a pointer variable.

component-selector

One of the following: an uparrow (↑), a record field selector followed by a record field identifier (for example, .fieldname), or an array index designator (for example, [n,m]).

Rules and Restrictions

- The effect of a type cast operation occurs exactly where it is specified and has no effect elsewhere.
- A variable or a selected record field may be cast to any type having the same storage allocation size as that of the variable's original type. For example, in the expression:

  ```
  integervar::recordtype.field1::boolean
  ```

  recordtype must have a size of 16 bits, and field1 must have a size of 8 bits, or 1 bit if recordtype is packed.
- An expression may be cast to a type of any size. If the expression is cast to a smaller type, the expression's value is truncated on the left. If the expression is cast to a larger type, the new type implies data in the high-order portion of the expression value that is the size difference between the old expression type and its new type. That data is undefined.

  When assigning an expression result to a variable, you should clear the destination variable before use if an expression result may have a smaller storage allocation requirement than the variable. Doing so guarantees that the unused high-order portion of the variable does not contain unpredictable data.

  Note

  When the destination variable in an assignment statement involving two variables is 16 bits, the compiler clears the high-order portion if the source variable is smaller than 16 bits.
- In some instances, a data object may be type cast in a context that requires word alignment, even though the data object's initial declaration may have assigned the object to a byte boundary. No compile-time error results, but a run-time odd address error may ensue. Make the correction by inserting a 1-byte filler in the object's declaration to force word alignment.
- Except where the allocation size of an operand is the same in both packed and unpacked contexts (see Appendix E), do not type cast an unpacked operand that appears in the same statement or expression as a packed operand; corrupt data will result. For instance, assume you wish to type cast an 8-bit unpacked record field to a type (such as a Boolean value) so you can interpret the 8-bit unpacked record field as a 1-bit field. If you do this in a statement that contains a packed Boolean operand, the result will be incorrect if this unpacked field contains data in its upper seven bits. The compiler will not detect this error.

Examples

1. This example shows you how to check a packed array of 16 Boolean flags quickly to see if any of the flags are set.

```
VAR
  Flags : PACKED ARRAY [1..16] OF BOOLEAN;
  Any_flags_set : BOOLEAN;

BEGIN
  Any_flags_set := Flags :: UNSIGNED <> 0;
END;
```

2. This example shows you how to examine the fifth bit of a byte to see if the fifth bit is set.

```
TYPE
  Bit_map = PACKED RECORD
              Bit0,Bit1,Bit2,Bit3,Bit4,Bit5,Bit6,Bit7 : 0..1;
            END;
VAR
  Bite : 0..255;
  Bitset : BOOLEAN;

BEGIN
  Bitset := Bite::Bit_map.Bit4::BOOLEAN;
END;
```

3. This example shows you how to use type casting as a convenient way to implement block assignments where type or size is not important.

```
TYPE
  fiftychartype = PACKED ARRAY [1..50] OF CHAR;
  tenchartype = PACKED ARRAY [1..10] OF CHAR;
VAR
  ten_chars : tenchartype;
  fifty_chars : fiftychartype;

BEGIN
  fifty_chars := (ten_chars)::fiftychartype;
  { ten bytes starting at the address denoted by "ten_chars"
    are moved to the start address of "fifty_chars"; the
    remaining forty bytes of "fifty_chars" are undefined }
END;
```

Chapter 4
The Declaration Section

The first two parts of a Pascal block are the heading and the declaration section. The heading specifies the program, module, procedure, and function or process name. The declaration section contains sections that declare labels, variables and their types, procedures, functions, and processes and that define constant identifiers and user-created types. Each section is introduced by an appropriate reserved word: CONST, LABEL, TYPE, VAR, PROCEDURE, FUNCTION, or PROCESS. A block need not include all those sections, and any number of those sections may appear. CONST, LABEL, TYPE, and VAR sections should appear first, in any order, followed by subprogram declarations. Subprogram declarations with either the EXTERNAL directive or the EXTERNAL attribute may be placed anywhere in the declaration section.

This chapter describes the constant declaration, the label declaration, type definition, and the variable declaration. See Chapter 6 for information on procedures, functions, and processes.

4.1 Constant Declaration

The constant declaration specifies identifiers to represent constant values.

Syntax

$$\text{CONST} \left\{ \text{constant-identifier} = \left\{ \begin{array}{l} \text{scalar-constant} \\ \text{structured-constant} \end{array} \right\} \right\} ; \ldots$$

constant-identifier
The identifier to be used as the name of the constant.

scalar-constant
An integer value, an unsigned value, a long integer, a real number, a string, a Boolean value, a value of an enumerated type, or the identifier (optionally signed) of another declared constant value.

structured-constant

A structured constant consisting of a record or array type identifier followed, in parentheses, by a list of constant values appropriate to the record or array type. See Section 3.2.2 for syntax and examples.

The value assigned to a constant identifier cannot be an expression or a set constructor; string constants must be enclosed in apostrophes. The use of constant identifiers makes a program easier to read and understand. In addition, if you need to change the value of a constant, you can modify the CONST declaration instead of changing each occurrence of the value in the program. That capability makes programs simpler to maintain and easier to transport.

Examples

1. The following CONST declaration specifies seven constant identifiers. YEAR, PI, and TINYD are numeric constants. MONTH and INITIAL represent string values. Both LIE and UNTRUTH are equal to the Boolean value FALSE.

```
CONST
  YEAR = 1979;
  MONTH = 'January';
  INITIAL = 'p';
  PI = 3.141592;
  TINYD = 1.7253E-10;
  LIE = FALSE;
  UNTRUTH = LIE;
```

2. The following CONST declaration specifies a constant array of type Block that contains all asterisk (*) characters.

```
TYPE
  Block = ARRAY [1..2, 1..3, 1..4] OF CHAR;

CONST
  Brick = Block (2 OF (3 OF (4 OF '*')));

VAR
  Corner_row_of_brick : ARRAY[1..4] OF char;
        .
        .
        .
  Corner_row_of_brick := brick[1,1];
        .
        .
```

3. You can reference the fields of a structured constant in the same way as the fields of a structured variable.

```
TYPE
  Codes = 1..255 ;
  Users = RECORD
            Last_name, First_name  : PACKED ARRAY [1..10] OF CHAR;
            Badge_number : INTEGER;
            Region_code : Codes;
          END;

CONST
  Default_Owner = Users ( '       Doe', '      John', 2324, 1 );
```

```
VAR
  Current_Owner : Users ;
BEGIN
    .
    .
    .
  Current_owner.Badge_number := Default_owner.Badge_number;
    .
    .
    .
END.
```

4.2 LABEL Declaration

The LABEL declaration establishes the numeric identifiers that are used to make statements accessible to a GOTO statement (see Section 5.5).

Syntax

LABEL { integer } ,...;

integer
A decimal integer between 0 and 9999. When you declare more than one label, you can specify them in any order.

A label can precede any statement in a program. You must use a colon (:) to separate the label from the statement. Each label must precede exactly one statement within the scope of its declaration.

Example

```
LABEL 0, 6656, 778, 4352;
```

This LABEL section specifies the four labels 0, 6656, 778, and 4352. The labels need not be specified in numeric order.

4.3 TYPE Definition

The TYPE definition introduces the name and set of values for a type.

Syntax

TYPE { type-identifier = ⟦ [{attribute} ,...] ⟧ ⟦ PACKED ⟧ type ; } ...

type-identifier
The identifier to be used as the name of the type.

attribute
Additional information about variables of this type, provided through the READONLY, STATIC, UNSAFE, VOLATILE, and WRITEONLY attributes. (See Chapter 10.)

PACKED

The structure to be stored in as few bits as is feasible. You can specify PACKED in type definitions for arrays, records, and sets. Though permitted, PACKED has no effect on file types. You may not specify PACKED with enumerated and subrange types or with type identifiers.

type

A type definition, which can be any of the predefined scalar or structured types, an enumerated type, a subrange, a pointer type, or the identifier of a previously defined type.

Application Notes

1. The size of a type definition may not exceed 65536 bytes. The size of a PACKED type definition may not exceed 65536 bits or 8192 bytes.
2. Specifying PACKED for a structure may significantly increase the size of the object code generated to access the structure's data elements.
3. Refer to Appendix H for further information on the efficient use of type definitions.

Example

The following TYPE section defines seven types with their identifiers. Both Entertainment and Days_of_Week are enumerated types. Hours_Worked is an array with five integer components. Salary and Pay are identical arrays of 50 real numbers each. Ptr_to_Hits is defined as a pointer to a record of type Hits, which has the five fields listed.

```
TYPE
  Entertainment = (Dinner, Movie, Theater, Concert);
  Days_of_Week = (Sun, Mon, Tues, Wed, Thurs, Fri, Sat);
  Hours_Worked = ARRAY [Mon..Fri] OF INTEGER;
  Salary = ARRAY [1..50] OF REAL;
  Pay = Salary;
  Ptr_to_Hits = ^Hits;
  Hits = RECORD
           Title, Artist, Composer : PACKED ARRAY [1..30] OF CHAR;
           Weeks_on_Chart : INTEGER;
           First_Version : BOOLEAN;
         END;
```

4.4 Variable Declaration

The variable declaration allocates a variable and associates it with an identifier and a type.

Syntax

$$\text{VAR} \left\{ \begin{array}{l} \{\text{variable-identifier}\} ,\ldots : [\![\; [\; \{\text{attribute}\} ,\ldots \;] \;]\!] \\ [\![\text{ PACKED }]\!] \text{ type ;} \end{array} \right\} \ldots$$

variable-identifier

The identifier to be used as the name of the variable.

attribute

Additional information about the variable, provided through the AT, EXTERNAL, GLOBAL, STATIC, READONLY, UNSAFE, VOLATILE, and WRITEONLY attributes. (See Chapter 10.)

PACKED

The structure to be stored in as few bits as is feasible. You can specify PACKED in type definitions for arrays, records, and sets. Though permitted, PACKED has no effect on file types. You may not specify PACKED with enumerated and subrange types or with type identifiers.

type

A type identifier or a type definition. The type can be any of the predefined scalar or structured types, an enumerated type, a subrange, a pointer type, or the identifier of a previously defined type.

Application Notes

1. The size of a variable declaration may not exceed 65536 bytes. The size of a PACKED variable declaration may not exceed 65536 bits or 8192 bytes.
2. Specifying PACKED for a structure may significantly increase the size of the object code generated to access the structure's data elements.
3. Refer to Appendix H for further information on the efficient use of type definitions.

Example

The following VAR section declares six variables and indicates the type of each. The types for variables Choice and Weekly_Hours are specified by type identifiers, defined in a previous TYPE definition section (not shown). The types for variables Answer, Rumor, Temp, Grade, and Status_Link are specified by type definitions.

```
VAR
  Choice : Entertainment;
  Answer, Rumor : BOOLEAN;
  Temp : INTEGER;
  Grade : 'A'..'D';
  Weekly_Hours : Hours_Worked;
  Status_Link : [GLOBAL,UNSAFE] UNSIGNED;
```

Chapter 5

Pascal Statements

Pascal provides statements to perform various actions within the program. This chapter presents reference information on each of the statements, in alphabetic order.

These statements can appear anywhere in the executable part of a Pascal program, procedure, function, or process. Pascal includes both simple and structured statements. The simple statements are the assignment and GOTO statements, the procedure call, and the process invocation. The compound, conditional, repetitive, and WITH statements are the structured statements. They enclose simple and structured statements that must be executed in order, repetitively, or when conditions are met. You can use a structured statement anywhere in the program that a simple statement is allowed. This manual uses the term statement to mean either a simple or a structured statement.

5.1 Assignment Statement

The assignment statement assigns the value of an expression to a variable.

Syntax

variable := expression

variable

The identifier of an array component, a file buffer variable, a function identifier, a field of a record, or a variable of any type except a file.

expression

A simple or structured constant value, variable name, set constructor, function reference, or expression.

A MicroPower/Pascal variable has no default value; a value must be assigned with the assignment statement. The expression on the right of the assignment operator (:=) establishes the value to be assigned to the variable on the left.

The compiler does not detect the occurrence of an undefined pointer on the left side of an assignment statement.

Examples

1. The variable X is assigned the value of 1.

   ```
   X := 1;
   ```

2. The value of the Boolean expression A <B is assigned to T.

   ```
   T := A<B;
   ```

3. The set Vowel_Set is assigned the set constructor shown. The base type of Vowel_Set must include the characters 'A', 'E', 'I', 'O', and 'U'.

   ```
   Vowel_Set := ['A', 'E', 'I', 'O', 'U'];
   ```

4. The first component of My_Array is assigned the sum of the seventh component of My_Array and the fourteenth component of Your_Array.

   ```
   My_Array[1] := My_Array[7] + Your_Array[14];
   ```

5. Assume that Awardrec and New_Winner are record variables of assignment-compatible types. This example assigns the value of each field of New_Winner to the corresponding field of Awardrec.

   ```
   Awardrec := New_Winner;
   ```

5.2 CASE Statement

The CASE statement causes one of several statements to be executed, depending on the value of an ordinal expression.

Syntax

```
CASE case-selector OF
     {case-label-list : statement } ;... [[;]]
                    [[OTHERWISE { statement } ;... [[;]]]]
END
```

case-selector

An expression that evaluates to any ordinal type except LONG_INTEGER.

case-label-list

One or more constants, separated by commas, of the same type as the case selector.

statement

A simple or a structured statement.

Each case-label-list element is associated with a statement that may be executed. The list contains the value of the case selector expression for which the system should execute the associated statement. You can specify the case labels in any order. Each case label can appear only once within any CASE statement but can appear in other CASE statements.

At run time, the system evaluates the case selector and chooses which statement to execute. If the value of the case selector expression does not appear in any case-label-list, the system executes the statement(s) in the OTHERWISE clause.

The case selector expression must match one of the case labels if you omit the OTHERWISE clause. When you enable the compiler's CHECK command option, an error message results if the CASE statement fails to select an executable statement. When you disable the CHECK option, the result is undefined if the CASE statement fails and you have omitted the OTHERWISE clause.

Note

An error results if you do not specify at least one case-label-list element and associated statement. This condition is not detected by the compiler.

Examples

1. At run time, the system evaluates AGE and executes one of the statements.

```
CASE Age OF
  5,6 : IF Birth_Month > Sep THEN Grade := 1 ELSE Grade := 0;
  7 : BEGIN
      Grade := 2;
      Reading_Skill := TRUE;
      END;
  8 : Grade := 3;
END;
```

If Age is not equal to 5, 6, 7, or 8, the program contains an error.

2. An OTHERWISE clause is added in this example.

```
CASE Age OF
  5,6 : IF Birth_Month > Sep THEN Grade := 1 ELSE Grade := 0;
  7 : BEGIN
      Grade := 2;
      Reading_Skill := TRUE;
      END;
  8 : GRADE := 3;
  OTHERWISE Grade := 0;
    Reading_Skill := FALSE;
END;
```

   If the value of Age is not 5, 6, 7, or 8, the value 0 is assigned to the variable Grade, and the value FALSE is assigned to the variable Reading_Skill.

3. This example assigns a value to Alpha_Flag, depending on the value of the character variable Alphabetic.

```
CASE Alphabetic OF
  'A','E','I','O','U' : Alpha_Flag := Vowel;
  'Y' : Alpha_Flag := Sometimes;
  OTHERWISE Alpha_Flag := Consonant;
END;
```

5.3 Compound Statement

The compound statement permits you to group Pascal statements for sequential execution as a single statement.

Syntax

```
BEGIN
   ⟦{statement} ;... ⟦;⟧⟧
END
```

statement
A simple or a structured statement.

You create a compound statement by using any combination of Pascal statements, including other compound statements. You must use semicolons to separate the individual statements within a compound statement. No semicolon is required—although it is allowed—between the last statement and the END delimiter. Pascal treats the entire compound statement as a single statement. Examples of compound statements appear throughout this chapter.

5.4 FOR Statement

The FOR statement specifies the repetitive execution of a statement based on the value of an automatically incremented or decremented control variable.

Syntax

FOR control-variable := initial-value $\left\{ \begin{array}{c} \text{TO} \\ \text{DOWNTO} \end{array} \right\}$ final-value DO statement

control-variable
The identifier of a variable of any ordinal type except LONG_INTEGER. The variable must be local to the program or subprogram block containing the FOR statement and must not be a component of a structure or the object of a pointer.

initial-value
An expression that is assignment compatible with the control variable.

final-value
An expression that is assignment compatible with the control variable.

statement
A simple or a structured statement.

The control variable, the initial value, and the final value must be of the same ordinal type. The statements within the repeat range must not change the value of the control variable, although they may interrogate it.

At run time, completion tests are performed before the statement is executed. In the TO form, if the value of the control variable is less than or equal to the final value, the loop is executed, and the control variable is incremented. When the value of the control variable is greater than the final value, execution of the loop is complete. In the DOWNTO form, if the value of the control variable is greater than or equal to the final value, the loop is executed, and the control variable is decremented. When the value of the control variable is less than the final value, execution of the loop is complete.

Because completion tests are performed before the statement is executed, some loops are never executed; for example:

```
FOR Control := N TO N+Q DO
  Week[N] := Week[N]+Netpay;
```

If the value of N+Q is less than the value of N (that is, if Q is negative) the loop is never executed.

Pascal begins execution of a FOR statement by assigning the value of the initial-value parameter to the variable specified by the control-variable parameter. During subsequent iterations, Pascal increments or decrements, as applicable, the control variable by units of the appropriate type. For numeric values, Pascal adds or subtracts 1 upon each iteration. For values of other types, the control variable takes on each successive value of the type. For example, a control variable of type 'A'..'Z' is incremented or decremented to the next character value each time the loop is executed.

The FOR loop terminates when the loop count is completed or a GOTO statement is encountered. The value of the control variable is left undefined and does not contain a valid value. You must therefore assign a new value to the control variable if you use it elsewhere in the program.

Examples

1. This FOR loop computes the sum of the components of Int_Array with index values from Lowbound through Highbound.

```
FOR N := Lowbound TO Highbound DO
  Sum := Sum + Int_Array[N];
```

2. The DOWNTO form is used here to print an inverted list of all the leap years in the nineteenth century.

```
FOR Year := 1899 DOWNTO 1800 DO
  IF (Year Mod 4) = 0 THEN
  WRITELN(Year:4,' IS A LEAP Year');
```

3. This example shows how you can nest FOR loops. For each value of I, the system steps through all 10 values of J and assigns the value 0 to the appropriate array component.

```
FOR I := 1 TO 10 DO
  FOR J := 1 TO 10 DO
  A[I,J] := 0;
```

4. This example combines structured statements. The inner FOR statement computes the number of hours each employee worked from Monday through Friday. The outer FOR statement resets hours to 40 for each employee and computes each person's pay as the product of wage and hours worked.

```
FOR Employee := 1 TO N DO
  BEGIN
    Hrs := 40;
    FOR Day := Mon TO Fri DO
      IF Sick[Employee,Day]
      THEN
        Hrs := Hrs-8;
    Pay[Employee] := Wage[Employee] * Hrs;
  END;
```

5.5 GOTO Statement

The GOTO statement causes an unconditional branch to a statement prefixed by a label.

Syntax

GOTO label

label

A statement label.

Upon execution of the GOTO statement, program control shifts to the statement with the specified label.

The GOTO statement must be within the scope of the label declaration. You cannot use a GOTO statement that is outside a structured statement to jump to a label that is within that structured statement. A GOTO statement within a routine can branch to a labeled statement in an enclosing block only if the labeled statement appears in the block's outermost level of nesting. That is, the labeled statement cannot occur within a structured statement.

Example

This example shows how you can use the GOTO statement to exit from a loop. The loop computes the sum (Invertsum) of the inverses of the components of Real_Array. If, however, one of the components is 0, the sum is set to 0, and the GOTO statement forces an exit from the loop.

```
FOR I := 1 TO 10 DO
  BEGIN
    IF Real_Array[I] = 0.0
    THEN
      BEGIN
        Result := 0.0;
        GOTO 10
      END;
    Result := Result + 1.0/Real_Array[I];
  END;

10: Invertsum:= Result;
    .
    .
    .
```

5.6 IF-THEN Statement

The IF-THEN statement causes the conditional execution of a statement.

Syntax

IF expression THEN [statement]

expression
A Boolean expression.

statement
A simple or a stuctured statement.

The statement is executed only if the value of the expression is TRUE. Otherwise, program control passes to the statement following the IF-THEN statement.

The THEN clause can specify a structured statement. However, if you use the compound statement, you must not place a semicolon between the words THEN and BEGIN—for example:

```
IF Day = Thurs THEN;     (* misplaced semicolon *)
BEGIN
statement
.
.
.
END;
```

As a result of the misplaced semicolon, the empty statement becomes the object of the THEN clause. In this example, the compound statement following the IF-THEN statement will be executed regardless of the value of Day.

Examples

1. If the value of the arithmetic expression is greater than 1000.0, a new value is assigned to the variable Answer.

```
IF ((X*37/Constant) + Factor) > 1000.0
THEN
  Answer := Answer - Factor;
```

2. If both relational expressions are true, D is assigned the value of A–C. Note that Pascal does not always evaluate all the terms of a Boolean expression if it can evaluate the entire expression based on the value of one term. Thus, if A is less than or equal to B, the expression B> C may not be evaluated.

```
IF (A>B) AND (B>C)
THEN
  D := A-C;
```

3. This example counts the number of J Smiths, prints each one's street address, and stores it in an array.

```
IF  (Name = 'Smith') AND (Initial = 'J')
THEN
  BEGIN
    Count := Count + 1;
    Smithadd[Count] := Address;
    WRITELN ('J Smith no. ',Count, ' Lives At ', Address);
  END;
```

4. If the value of the variable Day is Thurs, the FOR loop is executed, and values for Pay[i] are computed. If the value of Day is not Thurs, the FOR loop is not executed. Program control passes to the statement following the end of the loop.

```
IF Day = Thurs
THEN
  FOR I := 1 TO Max_Emp DO
    Pay[i] := Salary[i] * (1-Tax_Rate_Fica);
```

5.7 IF-THEN-ELSE Statement

The IF-THEN-ELSE statement is an extension of the IF-THEN statement and includes an alternative statement, the ELSE clause. The ELSE clause is executed if the test condition is false.

Syntax

IF expression THEN statement1 ELSE statement2

expression
A Boolean expression.

statement1
The simple or structured statement to be executed if the expression is true.

statement2
The simple or structured statement to be executed if the expression is false.

The objects of the THEN and ELSE clauses can be any simple or structured statement, including another IF-THEN or IF-THEN-ELSE statement. The ELSE clause is always paired with the closest IF-THEN statement—for example:

```
IF A=1 THEN
       IF B<>1 THEN C:=1
               ELSE D:=1;
```

By definition, Pascal interprets the statement above as if it included BEGIN and END delimiters, as follows:

```
IF A=1 THEN
       BEGIN
         IF B<>1 THEN C:=1
         ELSE D:=1;
       END;
```

The variable D is assigned the value 1 if both A and B are equal to 1. An ELSE clause to be executed if A is not equal to 1 would be placed as follows:

```
IF A=1 THEN
       IF B<>1 THEN C:=1
       ELSE D:=1
ELSE C:=0;
```

Examples

1. This example prints a different line of text, depending on the value of the Boolean variable Disease.

```
IF Disease
THEN
    WRITELN ('This person is sick.')
  ELSE
    WRITELN ('This person is healthy.');
```

2. If the value of Balance is negative, the compound statement is executed. The compound statement prints two lines of notification, adds a loan to Balance, and computes the amount of the bill for the loan. A zero or positive Balance results in a message stating that no loan was issued. The WRITELN procedure that prints the final balance is independent of the conditional statement and is always executed.

```
IF Balance < 0.0
  THEN
  BEGIN
    WRITELN ('Overdrawn by ',ABS(BALANCE));
    WRITELN ('Loan of ',Loan,' at ',Rate,' % automatically deposited');
    Balance := Balance + Loan;
    Bill_Amt := Loan * (1+Rate);
  END
  ELSE WRITELN ('No loan issued this month ');
  WRITELN ('Balance is ',Balance);
```

5.8 Procedure Call

A procedure call specifies the actual parameters to a procedure and executes the procedure. You may include a procedure call in any part of a program or subprogram where a statement can be used. See Chapter 6 for a complete description of procedures.

Syntax

procedure-identifier ⟦ ({actual-parameter} ,...) ⟧

procedure-identifier
The identifier of the procedure being called, as declared in the procedure's formal declaration.

actual-parameter
A constant, an expression, the identifier of a procedure or function, or a variable of an appropriate type. (Pascal passes actual parameters as described in Section 6.6.)

The procedure call associates a list of actual parameters with the formal parameters in the heading of the procedure declaration, then transfers control to the procedure.

The formal parameter list in the procedure declaration determines the possible contents and the order of the actual parameter list. The actual parameters must be compatible with the formal parameters. Depending on the types of the formal parameters, the actual parameters can be constants, variables, expressions, procedure names, or function names. An array name without an index in a parameter list refers to the entire array. You can specify the parameters in an order other than that implied by the formal parameter list by using the nonpositional syntax described in Section 6.6.1. You may omit parameters for which a default value or variable has been specified in the formal parameter list (see Section 6.6.2).

Examples

1. Suppose that you have the following procedure declaration:

```
TYPE
  T=ARRAY[1..10] OF INTEGER;
PROCEDURE Tollbooth
  (VAR Change: REAL;
       Toll: REAL;
       Lane: T);
```

   The following statement calls the procedure Tollbooth, specifying the variable Change, the real constant 0.25, and the array Lane.

```
Tollbooth (Change_X, 0.25, Lane_X);
```

2. Suppose that you have the following procedure declaration:

```
TYPE
  Status = (Pay, Collect, No_Owe);
VAR
  My_Amount, Rate, Income: REAL;
PROCEDURE Taxes
  (Real_Tax: REAL;
  VAR Amount_Withheld: REAL;
  Action: Status);
```

The following statement calls the procedure Taxes, with the expression Rate*Income, the variable My_Amount, and the identifier Pay as actual parameters.

```
Taxes (Rate*Income, My_Amount, Pay);
```

3. Suppose that you have the following procedure declaration:

```
TYPE
  Color_Code=(Red, Orange, Yellow, Green, Violet);
  Action_Str=PACKED ARRAY[1..10] OF CHAR;
  Status_Type=INTEGER;
PROCEDURE Check_Flag
  (Condition: Color_Code;
  Code: Action_Str;
  Status: Status_Type);
```

The following statement calls the procedure Check_Flag, using nonpositional syntax to specify the parameters 6, Green, and TEST.

```
Check_Flag ( Status := 6, Condition := Green,
Code := '      TEST' )
```

5.9 Process Invocation

A process invocation creates a dynamic process and specifies its actual parameters. You may include a process invocation in any part of a program or subprogram where a statement can be used. Although structurally similar to procedures, processes differ greatly from the other subprogram types in that processes execute concurrently and do not necessarily run to completion. See the *MicroPower/Pascal Run-Time Services Manual* for detailed information on processes and process relationships.

You invoke a process by specifying its identifier (given in the process's formal declaration) followed optionally by a parameter list enclosed in parentheses. The process invocation associates the actual parameters in the list with the formal parameters in the heading of the process declaration (see Chapter 6).

Each time the same process is invoked, a replication of the process is created using the data specified by the actual and predeclared parameters in the invocation statement.

The process that invokes a process is called the parent. When you invoke a process, you can specify the kind of dependency relationship that will exist between the parent and the process that the parent invokes. The invoked process can be either dependent or independent of the parent. If a process requires access to variables declared within the parent, you declare the process to be dependent (the default condition). If a process does not require access variables declared within the parent, you declare the process to be independent. When a process is dependent, the Pascal OTS will automatically preserve the heap space allocated for the parent processes' variables and OTS run-time structures until the dependent process exits. By creating independent processes, you make more efficient use of heap space since the invoked process does not require the presence of the parent process and the heap space allocated to the parent process.

The predeclared parameters DESC, NAME, PRIORITY, RELATION, STACK_SIZE, and STATUS establish the identification and environment for each invocation of a process.

Syntax

$$\text{process-identifier} \left[\!\!\left[\left(\begin{Bmatrix} \text{actual-parameter} \\ \text{predeclared-parameter} \end{Bmatrix} ,\ldots \right) \right]\!\!\right]$$

process-identifier
The identifier of the process being invoked, as declared in the process's formal declaration.

actual-parameter
A constant, an expression, the identifier of a procedure or function, or a variable of an appropriate type. (Pascal passes parameters as described in Section 6.6.)

predeclared-parameter
One or a combination of the predeclared parameters described below.

Predeclared Parameters

The compiler automatically declares the formal parameters DESC, NAME, PRIORITY, RELATION, STACK_SIZE, and STATUS for each process you create. Those parameters permit the kernel to obtain and to pass control information about a process.

Note

> Because the order in which the predeclared parameters are declared is not within your control, DIGITAL recommends that you use the nonpositional syntax described in Section 6.6.1 to avoid possible incompatibilities among the MicroPower/Pascal versions.

DESC := process-descriptor

The identifier of the descriptor variable to use for this invocation of the process. Process-descriptor must be a variable of predefined type PROCESS_DESC. That variable will be initialized with the process's identifier when the process is invoked. Thereafter, you may reference that variable in process-management requests (Chapter 12) to identify a process. If you do not specify a descriptor variable, the process can be referenced by name only. You should specify a new descriptor variable for each concurrent process.

NAME := process-name

The name of a particular invocation of a process. Process-name must be a 6-character string. If you use fewer than six printing characters, you must pad the string with trailing spaces. You may specify the name either as a string constant or as a variable that contains the string. Uppercase and lowercase versions of the same character are treated as unique. You may supply a default value for this parameter by specifying the NAME attribute in the process heading (see Section 10.2.14).

PRIORITY := process-priority

The execution priority of a process. Process-priority must be an integer value in the range 0 to 254 that specifies the priority. The highest priority, 255, can be specified only with the predeclared procedure CHANGE_PRIORITY (see Section 12.1). The default priority value for a process is the priority of the program or process that invoked it. You may change the default value for this parameter by specifying the PRIORITY attribute in the process heading (see Section 10.2.19).

$$\textbf{RELATION} := \left\{ \begin{array}{c} \textbf{DEPENDENT} \\ \textbf{INDEPENDENT} \end{array} \right\}$$

The relationship between the invoked process and the parent process—the process from which the invocation is issued—with regard to the accessibility and visibility of variable data.

- DEPENDENT (default)

  The invoked process is dependent on the parent process for access to intermediate-level nonstatic local variables or for access to variable data passed as parameters that is locally defined within the parent (see Rules below). With this option in effect, the Pascal OTS will not allow a parent process to exit before its dependent process exits.

- INDEPENDENT

  The invoked process is not dependent on the parent process either for access to intermediate-level nonstatic local variables or for access to variable data passed as parameters that is locally defined within the parent (see Rules below). With this option in effect, the Pascal OTS will allow the parent process to exit before an independent process exits.

STACK_SIZE := stack-size

The number of bytes of memory to allocate for a process's stack. The stack-size value you select must be a positive integer constant in the range 0 to 65,532, must be a multiple of 4, and must be less than the value specified for the DATA_SPACE attribute. You may change the default value, 400 bytes, for this parameter by specifying the STACK_SIZE attribute (see Section 10.2.22) in the process heading. The steps to follow for determining an appropriate STACK_SIZE value to specify are provided in the description of the STACK_SIZE attribute (see Section 10.2.22).

STATUS := status-record

The identifier of a variable of predefined record type EXC_STATUS that may receive an exception type and code. If you specify this parameter, the exception status, either success or error, that results from issuing the request is reported in that variable. Otherwise, an error causes the corresponding exception to be reported. The format of the exception record is described in Section 11.1.2.

Rules

a. The formal parameter list in the heading of the process declaration determines the possible contents and the order of the actual parameter list. The actual parameters must be compatible with the formal parameters. Depending on the types of the formal parameters, the actual parameters can be constants, variables, expressions, or routine identifiers.

b. You may omit parameters for which a default value has been specified in the formal parameter list (see Section 6.6.2).

c. If you invoke a named process so it is concurrent with a previous invocation of itself, you should supply new NAME and DESC parameters to avoid a conflict with those used in the first invocation. If you do not supply a new name, a run-time exception condition will occur. If you do not supply a new descriptor, the kernel will reinitialize the default descriptor and will use it as the descriptor for the current invocation of the process.

d. Unless the variable is statically allocated, the parent process should not declare a variable passed by VAR semantics to a process invoked with RELATION := INDEPENDENT.

Note

Statically allocated variables are created either when you declare them at the highest program level or when you use the AT, EXTERNAL, GLOBAL, or STATIC attributes.

e. A variable of the pointer type passed to a process invoked with RELATION := INDEPENDENT should not point to a data item declared in the parent process unless that data item is statically allocated.

f. A variable of a structured type passed by VAR or by value semantics to a process invoked with RELATION := INDEPENDENT should not contain components that contradict the requirements of Note e.

g. A parent process should not deallocate (DISPOSE procedure) a dynamic variable passed by VAR semantics to a process invoked with RELATION := INDEPENDENT.

h. A process invoked with RELATION := INDEPENDENT that is (lexically) declared within the parent process may access only those variables declared at the outermost (program) level or intermediate-level variables that are statically allocated.

i. If a process, invoked from within a routine, is passed local variable data by VAR semantics, the routine must not exit before the process terminates. Failure to observe this rule may result in erroneous data. Variable data passed to a process invoked in this way must be declared either at the outermost (program) level or be statically allocated.

Examples

1. Suppose that you have the following process declared:

```
[NAME ('Task_1'), PRIORITY (14), STACK_SIZE (1500)]
PROCESS Task (Inq_No: INTEGER);
```

The process begins running in response to the following invocation statement:

```
Task(I, PRIORITY:=32, STACK_SIZE:=2500, NAME:='Task_2', DESC:=Pdesc);
```

The process will have parameter value I, a stack size of 2500, a priority of 32, and a name of 'Task_2', and its descriptor will be stored in variable Pdesc.

2. The following example shows two invocations of the same process. In the first invocation, the process is given the default name 'Proc1 ', which was specified in the process heading by the NAME attribute. The second invocation of the process is named 'Proc2 '.

```
VAR Pname :  PACKED ARRAY[1..6] OF CHAR;

[NAME('Proc1 ')] PROCESS Task(I:INTEGER);
   BEGIN
     .
     .
     .
   END;
BEGIN
     .
     .
     .
  Task(I); (* This will have the default name 'Proc1 '. *)

  Pname:='Proc2 ';

  Task(I, NAME:=Pname); (* This will have the name 'Proc2 '. *)

END.
```

3. The following example shows two unnamed invocations of process P2, in which a descriptor is provided only for the second invocation.

```
VAR
  Pdsc, Newdsc : PROCESS_DESC;

[PRIORITY(27)] Process P2(I:INTEGER);
   BEGIN
     P2(I);                  (* The descriptor is disregarded. *)
     P2(I, Desc:= Newdsc);   (* The descriptor is put in *)
                             (* Newdsc. *)
   END;
```

Error Returns

See Section 11.2 for general information about error returns. The following exception codes may be returned:

ES$IPM	(type: SYSTEM_SERVICE)—Illegal parameter; illegal PRIORITY or STACK_SIZE parameter value
ES$NMK	(type: RESOURCE)—Insufficient space for kernel structure; the process could not be created
ES$SNI	(type: SYSTEM_SERVICE)—Structure name in use; a kernel structure already exists with the name you specified for the process

The request may return the following error, though not as a result of standard Pascal programming practice:

ES$IAD	(type: SYSTEM_SERVICE)—Invalid address; pointer to buffer or structure is odd or not in user address space

5.10 REPEAT Statement

The REPEAT statement groups one or more statements for execution until the specified condition is true.

Syntax

REPEAT { statement } ;... UNTIL expression

statement

A simple or a structured statement.

expression

A Boolean expression.

The format of the REPEAT statement lets you include several statements between the reserved words REPEAT and UNTIL without specifying them as a part of a compound statement. Because the expression is evaluated after the statements are executed, the REPEAT group is always executed at least once.

Example

Assume that the variable X is of type CHAR and that the variables Digit_Count, Digit_Sum, and Char_Count are integers. The example reads a character (X). If X is a digit, the count of digits is incremented by 1, and the sum of digits is increased by the value of X. The ORD function, described in Section 8.18, computes the value of X. If X is not a digit, the variable Char_Count is incremented by 1. The example continues processing characters until it reaches an end-of-line condition.

```
REPEAT
  READ(X);
  IF (X IN ['0'..'9'])
  THEN
    BEGIN
      Digit_Count := Digit_Count + 1;
      Digit_Sum := Digit_Sum + ORD(X) - ORD('0');
    END
  ELSE Char_Count := Char_Count+1;
UNTIL EOLN(INPUT);
```

5.11 WHILE Statement

The WHILE statement causes a statement to be executed while a specified condition is true.

Syntax

WHILE expression DO statement

expression
A Boolean expression.

statement
A simple or a structured statement.

The WHILE statement causes the statement following the word DO to be executed while the expression is true. Unlike the REPEAT statement, the WHILE statement controls the execution of only one statement. Hence, to repeatedly execute a group of statements, you must use a compound statement. Otherwise, Pascal repeats only the single statement immediately following the word DO.

The expression is evaluated before the statement is executed. If the expression is initially false, the statement is never executed. The repeated statement must change the value of the expression; otherwise, the result is an infinite loop.

Examples

1. This statement skips to the end of the text file FILE1.

```
WHILE NOT EOF (FILE1) DO
  READLN (FILE1);
```

2. This example reads an input character from the current line on the terminal. If the character is not a digit or a letter, the error count (ERR) is incremented by 1.

```
WHILE NOT EOLN(INPUT) DO
  BEGIN
    READ(X);
    IF NOT (X IN ['A'..'Z','a'..'z','0'..'9'])
    THEN
      Err := Err + 1;
  END;
```

3. After initializing Sum to 0, this program fragment repeatedly calculates a student's average test score. If the average score falls below 90, the calculations cease, and the system prints an informational message. If the average never falls below 90, calculations continue until Ntests is greater than Maxtests, and no message is printed.

```
Sum := 0;
Ntests := 1;
Avg := 100;

WHILE  (Avg >= 90) AND (Ntests <= Maxtests) DO
  BEGIN
    Sum := Sum + Test [NTests];
    Avg := Sum DIV Ntests;
    Ntests := Ntests +1;
  END;
IF Avg < 90 THEN
  WRITELN ('Your average dropped below 90 as of test ', Ntests:5);
```

5.12 WITH Statement

The WITH statement provides abbreviated notation for references to fields of a record.

Syntax

WITH { record-variable } ,... DO statement

record-variable

The identifier of the record variable.

statement

A simple or a structured statement.

The WITH statement allows you to refer to the fields of a record directly instead of using the record.fieldname syntax. In effect, the WITH statement opens the scope of the field identifiers, allowing you to use them as you would use variable identifiers.

Specifying more than one record variable has the same effect as nesting WITH statements. Thus, the following two statements are equivalent:

```
WITH Cat, Dog DO
  Bills := Bills + Catvet + Dogvet;
```

and

```
WITH Cat DO
  WITH Dog DO
    Bills := Bills + Catvet + Dogvet;
```

If the record Dog is nested within the record Cat, you must specify Cat before Dog. The names must appear in the order of their declaration.

Examples

1. This statement tests the value of the field Taxes.Net and sets Taxes.Itemized to TRUE if Taxes.Net is less than 10000.0.

```
VAR
  Taxes : RECORD
            Gross : REAL;
            Net : REAL;
            Bracket : REAL;
            Itemized : BOOLEAN;
            Paid : REAL;
          END;
    .
    .
    .
WITH Taxes DO
  IF Net < 10000.0 THEN Itemized := TRUE;
```

2. The program segment in this example shows how you can use the WITH statement to assign values to the fields of a record. The WITH statement specifies the names of the record variables Hosp and Birthdate. The record names must be in order; that is, Hosp must precede Birthdate. The assignment statements need specify only the field names—for example, Patient instead of Hosp.Patient and Month instead of Hosp.Birthdate.Month.

```
TYPE
  Name = PACKED ARRAY [1..20] OF CHAR;
  Date = RECORD
           Month : (Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec);
           Day : 1..31;
           Year : INTEGER;
         END;

VAR
  Hosp : RECORD
           Patient : Name;
           Birthdate : Date;
           Age : INTEGER;
         END;
    .
    .
    .
WITH Hosp, Birthdate DO
  BEGIN
    Patient := 'Thomas Jefferson    ';
    Month := Apr;
    Age := 236;
  END;
```

Chapter 6

Subprograms: Procedures, Functions, and Processes

You can usually break down a complex problem into a collection of simpler subproblems. MicroPower/Pascal's subprogram constructs let you segment programs into procedures, functions, and processes. You can thus isolate the individual tasks that the main program is to accomplish. By developing and debugging each subprogram independently, you increase the probability that the entire program will execute successfully.

6.1 Concept of Subprograms

A subprogram is an identifiable entity consisting of definitions, declarations, and executable statements that execute as a group. The two categories of subprograms are routines, which consist of procedures and functions, and processes. Routines and processes are structurally similar but semantically different.

You can either include subprograms in the main program or compile them as separate modules. A MicroPower/Pascal program can include subprograms of the following categories:

- User-created internal subprograms
- User-created external subprograms written in Pascal
- User-created external subprograms written in MACRO–11 assembly language
- Predeclared subprograms (see Chapters 8, 9, and 11 to 17)

6.1.1 Routines: Procedures and Functions

Both procedures and functions perform a set of closely related program steps but differ in the context in which they are used. A procedure performs a task that does not necessarily require or return data and is activated by an executable statement known as a procedure call, described in Section 5.8. A function computes and returns a value of its declared result type to the calling program or subprogram. You may call a function by naming it as an operand within an expression where its result type is allowed (see Chapter 3). Routines execute in strict sequence; they do not execute independently of other parts of a program.

Pascal supplies many predeclared routines that perform commonly used operations, including arithmetic operations, input and output services, and real-time requests from the MicroPower/Pascal kernel.

6.1.2 Processes

A process is similar in structure to a routine; rather than executing in sequence, however, a process executes concurrently—logically in parallel—with other processes in a given application program. The process construct lets you decompose an otherwise monolithic, sequential program into autonomous subprograms that are scheduled independently when triggered by appropriate events. A process provides a simpler conceptual approach to solving real-time problems than one that uses sequential programming techniques. You invoke, or create, a process by using an executable statement known as a process invocation, described in Section 5.9.

The real-time programming requests, described in Part Two, provide the process synchronization and communication services necessary for effective concurrent processing. The *MicroPower/Pascal Run-Time Services Manual* describes processes and concurrent processing in detail.

6.1.3 Subprogram Structure

A subprogram consists of a heading and a block. The heading identifies the subprogram and may include a list of identifiers that the subprogram will use to exchange data with the calling program or subprogram. The body consists of either a block or a directive. A block contains a declaration section and an executable section.

You may create up to 14 levels of nested subprogram declarations. Declarations that are global or external, however, must appear at the outermost, or top, program or module level—a technique called subprogram nesting. A directive supplies information about forward-declared and external subprograms. Subprograms must be declared in a declaration section of a main program, module, or another subprogram before they can be called. A procedure call, function identifier, or process invocation statement in the executable section of a main program or another subprogram causes a subprogram to run.

Subprograms exchange data with the main program and with each other by means of function results and identifiers called parameters. The formal parameters (parameters used within the subprogram block) must be listed in the subprogram heading. At execution, each formal parameter corresponds to an actual parameter provided in the subprogram call. You can invoke a subprogram several times with different actual parameters. Pascal checks every call to a subprogram to ensure that the types of the actual and formal parameters are compatible.

The scope of an identifier is the part of the program in which the identifier is accessible. In Pascal, the scope of an identifier—which represents a constant, variable, type, procedure, process, function, or a label—is the block in which the identifier is defined or declared, minus any nested blocks that redeclare the same identifier or label. The declaration section in the main program block introduces identifiers that are accessible in the main program and in all nested subprograms. The declaration sections in subprogram blocks specify local identifiers. You can use a local identifier in the subprogram that contains the local identifier's declaration and in all nested subprograms. You can redeclare in a subprogram an identifier that has been declared in an outer block; the identifier always refers to the declaration of most limited scope.

6.2 Subprogram Declarations

To declare a subprogram, you supply the subprogram's heading and either its block or a directive in the declaration section of a main program, a module, or another subprogram. Normally, you must declare subprograms before you can refer to them within a program. The FORWARD directive, described in Section 6.5.1, lets you escape that restriction. The subprogram declaration provides all the information necessary to determine whether the actual parameters in a call to the subprogram can legally be passed to the formal parameters in the declaration.

Syntax

$$\left\{\begin{array}{l}\text{procedure-heading}\\ \text{function-heading}\\ \text{process-heading}\end{array}\right\}\left\{\begin{array}{c}\text{directive}\\ \\ \text{block}\end{array}\right\};$$

procedure-heading
A procedure heading (see Section 6.2.1).

function-heading
A function heading (see Section 6.2.2).

process-heading
A process heading (see Section 6.2.3).

directive
The FORWARD, SEQ11, or EXTERNAL directives (see Section 6.5).

block
Defined as follows:

$$\left\{\begin{array}{l}\text{constant-declaration}\\ \text{label-declaration}\\ \text{type-definition}\\ \text{variable-declaration}\\ \text{subprogram-declaration}\end{array}\right\}\text{... BEGIN } [\![\text{ \{ statement \} ;... }]\!] \text{ END}$$

constant-declaration
A constant declaration, as defined in Section 4.1.

label-declaration
A label declaration, as defined in Section 4.2.

type-definition
A type definition, as defined in Section 4.3.

variable-declaration
A variable declaration, as defined in Section 4.4.

subprogram-declaration
A subprogram declaration, as defined above.

statement
One of the MicroPower/Pascal statements, described in Chapter 5.

6.2.1 Procedure Heading Syntax

The following syntax diagram defines the procedure heading:

Syntax

⟦ [{ attribute } ,...] ⟧ PROCEDURE procedure-identifier
⟦ (formal-parameter-list) ⟧ ;

attribute
GLOBAL, EXTERNAL, INITIALIZE, and TERMINATE attributes. The GLOBAL and EXTERNAL attributes may be used only on procedures declared at the outermost level of a program or module. (See Chapter 10 for more information on attributes.)

procedure-identifier
The identifier to be used when calling the procedure.

formal-parameter-list
The identifiers and types of the formal parameters (see Section 6.3).

Example

```
PROCEDURE Put_Data
          (VAR Buffer: [UNSAFE] Big;
               Byte_Count:  INTEGER);
```

6.2.2 Function Heading Syntax

The following syntax diagram defines the function heading:

Syntax

⟦ [{ attribute } ,...] ⟧ FUNCTION function-identifier
⟦ (formal-parameter-list) ⟧ : result-type-identifier ;

attribute

GLOBAL and EXTERNAL attributes, which may be used only on functions declared at the outermost program or module level. (See Chapter 10 for more information on attributes.)

function-identifier

The identifier to be used when invoking the function and the name of the function result.

formal-parameter-list

The identifiers and types of the formal parameters (see Section 6.3).

result-type-identifier

The data type of the value returned by the function.

Example

```
FUNCTION Count_substr
  (VAR String: [READONLY] PACKED ARRAY
  [Ls..Us: INTEGER] OF CHAR; VAR Key: [READONLY] PACKED ARRAY
  [Lk..Uk: INTEGER] OF CHAR): INTEGER;
```

6.2.3 Process Heading Syntax

The following syntax diagram defines the process heading:

Syntax

⟦ [{ attribute } ,...] ⟧ PROCESS process-identifier
  ⟦ (formal-parameter-list) ⟧ ;

attribute

CONTEXT, GROUP, NAME, PRIORITY, STACK_SIZE, GLOBAL, and EXTERNAL attributes. The GLOBAL and EXTERNAL attributes may be used only on processes declared at the outermost level of a program or module. (See Chapter 10 for more information on attributes.)

process-identifier

The identifier to be used when invoking the process.

formal-parameter-list

The identifiers and types of the formal parameters (see Section 6.3).

Example

```
[PRIORITY(10), STACK_SIZE(100), NAME ('DRIVER')] PROCESS Driver;
```

6.3 Formal Parameters

A formal parameter establishes a logical data path between a subprogram and its caller and specifies the characteristics of that data. A formal parameter may be used to provide a data value to a subprogram or to return a value to the calling program or subprogram. The function result is a special case of a parameter and returns only data. Parameters used purely to supply data to a subprogram may be specified as value parameters. Parameters that return values to the caller are specified as variable (VAR) parameters. Parameters that allow a calling block to pass a routine are specified as procedure and function parameters.

MicroPower/Pascal lets you declare formal parameters that can accept arrays of different sizes. By using the conformant array syntax, a subprogram can process arrays of different sizes within specified bounds.

You can also associate a default value with a formal value or variable parameter when you declare it, so you need only specify an actual parameter if you want to pass a different value or address (see Section 6.6.2).

6.3.1 Formal Parameter List

A formal parameter list is composed of parameter sections. A parameter section introduces one or more formal parameter identifiers and indicates how they will be interpreted within the subprogram.

Syntax

$$\left\{ \begin{array}{l} \text{value-parameter-section} \\ \text{variable-parameter-section} \\ \text{procedure-parameter-section} \\ \text{function-parameter-section} \end{array} \right\} ;\ldots$$

value-parameter-section
Parameters to be used only to input data (see Section 6.3.2).

variable-parameter-section
Parameters to be used for both input and output (see Section 6.3.3).

procedure-parameter-section
Parameters that identify a procedure and its parameters to the subprogram (see Section 6.3.4).

function-parameter-section
Parameters that identify a function and its parameters to the subprogram (see Section 6.3.4).

The following subsections describe the semantics of parameter passing in Pascal and the use of each kind of parameter.

6.3.2 Value Parameters

By the rules of value semantics, a formal value parameter acts like a local variable within the called subprogram. During execution, the value of an actual parameter is passed to the called subprogram to initialize the formal parameter. When control returns to the calling block, the formal parameter value is not retained. If the called subprogram assigns a new value to the formal parameter, the change is therefore not reflected in the calling block.

You may specify a default for a value parameter to omit commonly used parameters from an actual parameter list.

Syntax

{ identifier } ,... : ⟦ [{attribute} ,...] ⟧
  type-identifier ⟦ := default-value ⟧

identifier

The name of a formal parameter. Multiple identifiers must be separated with commas.

attribute

READONLY, UNSAFE, and VOLATILE attributes. (See Chapter 10 for more information on attributes.)

type-identifier

The type identifier of the parameters in this section.

default-value

A variable identifier or a constant for the parameter if no actual parameter is provided when the subprogram is invoked (see Section 6.6.2).

Rules and Defaults

- A default value for a value parameter declaration is used in place of the actual parameter value when that parameter is omitted from the actual parameter list. If you specify a variable identifier as the default, the contents of that variable will be used. A NIL default is interpreted as having the value 0.
- When a formal value parameter has the UNSAFE attribute, the types of the actual parameters passed to it are not checked for compatibility; their allocation sizes, however, must be the same.
- A variable specified as a default must have been declared at the outermost program or module level prior to this reference.
- The type of a constant or variable specified as a default must be identical to that of its corresponding formal parameter.
- The NIL default may be used only with a formal parameter that is of a pointer type.

Examples

1. This example shows how you can disable type checking on actual parameter values.

   ```
   PROCEDURE P1(X: [UNSAFE] INTEGER := 0);
   ```

   The procedure P1 has one parameter, X, of type INTEGER. The UNSAFE attribute disables type checking on actual parameter values. If no actual parameter is specified when P1 is called, a parameter value of 0 is assumed.

2. This example demonstrates the use of default value parameters.

```
TYPE
  iptr = ^INTEGER;
VAR
  i : INTEGER;
  p : iptr;
PROCEDURE z ( y : INTEGER := i;
              x : iptr := p;
              w : iptr := NIL );
BEGIN
END;
BEGIN
  z;          { use defaults for y, x, and w }
  z(i, ,p);   { use default for x }
  z(w := p);  { use defaults for y and x }
  z(, ,p);    { use defaults for y and x }
END.
```

6.3.3 Variable (VAR) Parameters

Pascal uses variable semantics to pass data to a formal parameter that is preceded by the reserved word VAR. Such a parameter is called a formal VAR parameter. By the rules of variable semantics, the formal variable parameter represents the address of a variable in the called subprogram. The subprogram accesses the actual variable directly to obtain its value rather than accessing a copy of it, as with value semantics. Thus, the subprogram can assign a new value to the formal parameter during execution, and the changed value will be reflected immediately in the calling block.

You may specify a default for a VAR parameter to omit a commonly used parameter from the actual parameter list. The default denotes the address of the specified variable.

Syntax

```
    { {identifier} ,... : ⟦ [ { attribute } ,...] ⟧                   }
VAR {                     type-identifier ⟦ := default-address ⟧      }
    {                     identifier : conformant-array               }
```

identifier

The name of a formal parameter. Multiple identifiers must be separated with commas.

attribute

READONLY, UNSAFE, VOLATILE, and WRITEONLY attributes. (See Chapter 10 for more information on attributes.)

type-identifier

The type identifier of the parameters in this section.

default-address

The default for the parameter (a variable identifier, an unsigned integer, or NIL) if no actual parameter is provided when the subprogram is invoked (see Section 6.6.2). The NIL default is interpreted as an unsigned integer of 0; that is, address 0.

conformant-array

Declaration of the formal parameter as a conformant array (see Section 6.3.3.1).

Rules and Defaults

- A default for a VAR parameter declaration is used in place of the actual parameter when that parameter is omitted from the actual parameter list. If you specify a variable identifier as the default, the address of that variable will be used. If you specify an unsigned integer, that value is taken as an address.
- A variable specified as a default parameter must have been declared at the outermost program or module level prior to this reference.
- The type of a variable specified as a default parameter must be identical to that of its corresponding formal parameter definition.
- An unsigned integer specified as a default parameter is not checked for type identity with its corresponding formal parameter.

Rules for effecting the use of default parameters in actual parameter syntax are provided in Section 6.6.2.

Application Notes

Because no copy is made of the actual VAR parameter, you can save storage space by using formal VAR parameters instead of value parameters. This technique can be especially helpful when you are passing actual parameters that require large amounts of storage space. However, to use a VAR parameter as an efficient substitute for a value parameter, you:

- Must not modify the actual parameter
- Should not refer to the actual parameter by more than one name within the same block (for example, by reference to a program-level variable)

Examples

1. Two examples of VAR declaration sections follow:

```
VAR A : List;
VAR Valid : BOOLEAN;
```

2. The following example illustrates how passing a large array to a formal VAR parameter differs from passing it to a value parameter:

```
TYPE
  Big_Array = ARRAY [0..1000] OF REAL;
PROCEDURE Reverse
  (VAR In_Array, Out_Array : Big_Array);
  VAR
    I, J : INTEGER;
  BEGIN
    J := 0;
    FOR I := 1000 DOWNTO 0 DO
      BEGIN
        Out_Array [I] := In_Array [J];
        J := J + 1;
      END;
  END;
VAR
  A1, A2 : Big_Array;
      .
      .
      .
Reverse (A1, A2);       (* Would execute successfully *)
Reverse (A1, A1);       (* Would fail *)
```

The second call to the Reverse procedure would fail to execute as you expect, because you are using the same array for both input and output. Since A1 is being passed to a VAR parameter, the procedure accesses A1 directly and modifies the input values as the procedure writes the values of the formal parameter into A1.

3. This example demonstrates how you can use default VAR parameters.

```
TYPE
  iptr = ^INTEGER;
VAR
  i : INTEGER;
  p : iptr;
PROCEDURE z ( VAR y : INTEGER := i;
              VAR x : iptr := p;
              VAR v : iptr := %O'10000' );
BEGIN
END;
BEGIN
  z;           { use defaults for y, x, and v }
  z(i, ,p);    { use default for x }
  z(i,p);      { use default for v }
  z(v := p);   { use defaults for y and x }
  z(, ,p);     { use defaults for y and x }
END.
```

6.3.3.1 Conformant Arrays

Some programming applications require general purpose subprograms that can process arrays or character strings with different bounds. For example, you could write a procedure that finds the minimum, maximum, and average of the components of a 1-dimensional array of integers. Similarly, you could write a function that returns the number of times one string occurs in another. Your subprogram would then treat the formal parameter as though its bounds were those of the actual. Unfortunately, since two arrays of different sizes are not of the same type, you could not define the type of a formal parameter that would accept both arrays.

A conformant array represents a collection of array types having the same component type but different dimension bounds. The bounds of an actual parameter are available within the subprogram through additional identifiers declared in the formal conformant array. A conformant array can be specified only in a variable-parameter section of a formal parameter list.

Syntax

Form 1

```
⟦ [ {attribute} ,... ] ⟧
    ARRAY [ {lower-bound-identifier..upper-bound-identifier :
                                        ┌ type-identifier    ┐
         index-type-identifier} ;... ] OF {                    }
                                        └ conformant-array   ┘
```

Form 2

```
⟦ [ {attribute} ,... ] ⟧
    PACKED ARRAY [lower-bound-identifier..upper-bound-identifier :
       index-type-identifier] OF CHAR
```

attribute

READONLY, VOLATILE, and WRITEONLY attributes. (See Chapter 10 for more information on attributes.)

lower-bound-identifier

The lower bound of the conformant array's index.

upper-bound-identifier

The upper bound of the conformant array's index.

index-type-identifier

The ordinal type of the index.

type-identifier

The type of the array components.

Rules and Defaults

- You must use type identifiers to specify the type of the index; they may be of any ordinal type.
- The upper- and lower-bound identifiers are implicitly declared as READONLY value parameters and may be referenced only within the subprogram block.
- Unless the conformant array is packed, the component may be either a type identifier or another conformant array.

Note

Except for PACKED ARRAY OF CHAR, MicroPower/Pascal does not allow packed conformant arrays.

- Two conformant arrays are compatible if both their index types and components are compatible.

Examples

1. This example shows a 1-dimensional conformant array.

```
PROCEDURE Sub_string
  (VAR A: PACKED ARRAY [I..J: INTEGER] OF CHAR;
  VAR B: PACKED ARRAY [L..M: INTEGER] OF CHAR;
  Start, Len: INTEGER);

  VAR
    Temp: INTEGER;

  BEGIN
    IF (I > Start)
      OR (J < Start + Len)
      OR (L > Start)
      OR (M < Start + Len)
    THEN WRITELN('?Bad strings')
    ELSE
      BEGIN
        FOR Temp := Start TO Start + Len DO
          A[I + Temp - Start] := B[Temp];
        FOR Temp := I + Len TO J DO A[Temp] := ' ';
      END;
  END (* Sub_string *) ;
```

The string specified by A is filled with a number of characters, specified by Len, from the string specified by B, beginning at Start. A is filled with spaces after the substring is placed in it. The bounds of the arrays are checked to make sure that the substring operation requested is legal.

2. A conformant array can have more than one dimension, as in this example.

```
TYPE
  Level_Range = 1..6;
  Nclasses = 1..8;
  Nstudents = 1..40;
  Names = PACKED ARRAY [1..35] OF CHAR;
  .
  .
  .
PROCEDURE Kid_Count
  (VAR School :
  ARRAY [Grade_Low..Grade_High : Level_Range;
    Units_Low..Units_High : Nclasses;
    Pupils_Min..Pupils_Max : Nstudents]
  OF Names);
```

This example declares School as a 3-dimensional conformant array parameter. Each array passed to School might contain the names of all the students in a particular elementary school. The indexes of the array denote the number of grades in the school, the number of classes at each grade level, and the number of students in each class.

3. To pass character strings, which are interpreted as constants, as conformant array parameters, the corresponding formal parameter must be declared as a VAR PACKED parameter with the READONLY attribute (see Sections 6.6.4 and 10.2.21). The reason is that conformant arrays must be declared as VAR parameters, and character strings passed as VAR parameters must be declared READONLY. The following example shows this:

```
FUNCTION Search  (* Return TRUE if String_A *)
                 (* occurs in String_B *)
    (VAR String_A:[READONLY]PACKED ARRAY [J..K:INTEGER]
       OF CHAR;
     VAR String_B:[READONLY]PACKED ARRAY [M..N:INTEGER]
       OF CHAR): BOOLEAN;
  VAR
    I, Count : INTEGER;
    Found,  Match : BOOLEAN;
  BEGIN
    I := 0;
    Found := FALSE;
    WHILE (NOT Found) AND (I <= (N - M)) DO
      BEGIN
        Count := 0;    Match := TRUE;
        WHILE Match AND (Count <= (K - J)) AND
          (Count + I <= (N - M)) DO
          BEGIN
            IF NOT ( String_A [ J + Count ] =
              String_B [ M + I + Count ] )
            THEN
              Match := FALSE;
              Count := Count + 1;
          END;
        IF Match AND ( Count = (K - J + 1) )
          THEN Found := TRUE;
        I := I + 1;
      END;
    Search := Found;
  END;
BEGIN (* Main Program *)
```

```
   .
Found := Search ('keyword', Some_character_array);
   .
   .
Found := Search ('abc','aababbabcabd');
   .
   .
END.
```

6.3.4 Procedure and Function Parameters

Formal procedure and function parameter semantics allow you to declare a formal parameter that is a procedure or a function. Thus, the calling block can pass a procedure or a function to a subprogram.

Syntax

$$\left\{ \begin{array}{l} \text{procedure-heading} \\ \text{function-heading} \end{array} \right\}$$

procedure-heading

The procedure heading to be passed by this parameter (see Section 6.2.1).

function-heading

The function heading to be passed by this parameter (see Section 6.2.2).

The identifiers listed in the formal parameter list of a formal procedure or function parameter are not accessible outside the routine declaration. The identifiers merely indicate the number and types of actual parameters necessary. You refer to those identifiers only when you use nonpositional syntax to pass the routine parameter (see Section 6.6.1).

Examples

1. This parameter includes a function name along with its own formal parameter list, result type, and value parameter.

   ```
   (FUNCTION Operation (Left, Right : REAL) : REAL;
   Result : REAL)
   ```

2. This specification allows you to pass a procedure to the subprogram declared with this formal parameter list.

   ```
   (PROCEDURE Display_Status (Error_Code : INTEGER));
   ```

6.3.5 Predefined Process Parameters

In addition to any formal parameters that you may declare, the compiler automatically declares the formal parameters DESC, NAME, PRIORITY, RELATION, STACK_SIZE, and STATUS for each process. Those parameters allow you to specify the name, descriptor variable, execution priority, stack size, and error status for each invocation of a process (see Section 5.9). You may specify default values for the NAME, PRIORITY, and STACK_SIZE parameters when you declare a process by using their like-named attributes NAME, PRIORITY, and STACK_SIZE, described in Chapter 10.

6.4 Subprogram Blocks and Scope of Identifiers

A subprogram block, like a program block, contains a declaration section and an executable section. The declaration section defines labels and identifiers for constants, types, variables, procedures, processes, and functions that are available within the block. An identifier defined in the declaration section may be used in subsequent declarations and definitions. The labels and identifiers declared in the block are local to the subprogram and are unknown outside its scope.

By default, all local variables are automatically allocated and deallocated; that is, the system does not retain the values of local variables once a subprogram completes execution. Because subprogram-level variables are automatically allocated, each call to a subprogram creates space for all local variables. You can therefore call a subprogram recursively without affecting the values held by the variables at prior activations of the subprogram. To preserve the value of a local variable (not the copy) from one invocation to the next, you must declare the local variable with the STATIC attribute (see Section 10.2.23).

The executable section of the block contains the statements that perform the actions of the subprogram. You can cause an exit from a subprogram block by a GOTO statement to a label outside the block of a routine or in the case of a process, by issuing a STOP real-time programming request.

6.4.1 Scope of Identifiers

In Pascal, the scope defines the legal limits of an identifier's accessibility. The scope of an identifier extends from the point at which its declaration or definition appears to the end of the block, minus any nested blocks that redeclare or redefine the identifier. The existence of scope rules helps limit the declaration or definition of an identifier to that part of the program in which it is used. You can take advantage of scope rules to use an identifier more than once within a program and give it different meanings. You should, however, limit the redefinition of identifiers to very short names, such as I, J, and X, to avoid confusion. The following rules of scope apply to Pascal identifiers:

- A previously declared identifier can be redeclared by a nested block.
- Identifiers declared in the main program block are accessible at all levels of the program; that is, their scope is the entire program.
- A procedure identifier can be redeclared within its own declaration section.
- A function identifier can also be redeclared except in the outermost subprogram-level declaration section. Because a function identifier must have a value assigned to it, it can be redeclared only in a nested subprogram.
- Formal parameter names follow the same rules of scope as identifiers declared within the block. A formal parameter name can be redeclared only in a nested subprogram.
- Label definitions follow rules of scope similar to those of identifiers. The scope of a label is the block in which it is declared, minus any nested blocks that redefine the label number. You can therefore transfer control from one block to an enclosing block, but you must follow certain restrictions, as outlined in Section 5.5.

Caution

If, within the block of a routine (procedure or function), you declare a process that accesses data from higher-level local (nonstatic) variables, you must ensure that the routine does not exit before the process terminates. Otherwise, the process should access only data that is static, to avoid erroneous results. Statically allocated variables are created either when you declare them at the outermost level of a program or module or when you use the AT, EXTERNAL, GLOBAL, or STATIC attributes.

Example

This example shows the scope of identifiers that appear in several blocks in a program.

```
VAR A, B : INTEGER;
.
.
.
PROCEDURE Level1a (Z, Y : INTEGER);
  TYPE C = ARRAY [1..35] OF CHAR;
  VAR D, E : C;
  .
  .
  .
  END; (* end procedure Level1a *)
PROCEDURE Level1b (V, U : CHAR; VAR T : INTEGER);
  .
  .
  .
  FUNCTION Level2:REAL;
    VAR B : BOOLEAN;
    .
    .
    .
    END; (* end function Level2 *)
  .
  .
  .
  END; (* end procedure Level1b *)
```

Because of Pascal's scope rules, the following statements about the identifiers declared in the example are true:

- Variables A and B are accessible everywhere in the program, and except in function Level2, which redeclares B as a BOOLEAN variable, they are treated as integers.
- Type identifier C and variables D and E are declared in procedure Level1a and are accessible in that block. The scope of C, D, and E, however, does not include the part of the program that is outside the declaring procedure. You could not, for example, use the variable E in procedure Level1b, because that block is outside the scope of the identifier E.
- Function Level2 redeclares the identifier B so it is a BOOLEAN variable rather than an integer. Inside Level2, B is treated as BOOLEAN, but elsewhere in the program, B is still interpreted as an integer. You may not redeclare B within the scope of the main program block, because B has already been declared there as an integer.

- The identifier Level1a is declared as a procedure name at the outermost level of the program. Level1a could have been redeclared in its own declaration section in addition to the procedure's local identifiers C, D, and E (which were already declared).
- The identifier Level2 is declared as a function name within procedure Level1b. Level2 cannot be redeclared within its own declaration section but could be redeclared within any nested blocks.
- The formal parameters V, U, and T in procedure Level1b cannot be redeclared as local identifiers within that procedure but could be redeclared within the nested block of function Level2.

6.4.2 Function Result

Within a function block, the function identifier acts much like a variable and is called the function result. Any attributes associated with the function result apply only within the function block. When the function is called, its result is undefined. A function block should include at least one statement (for every potential path through the code) that assigns a value of the result type to the function identifier. The function result is the last value assigned to the function identifier. When the function finishes executing, its result is returned to the calling block.

Note

The compiler checks that at least one assignment to the function result is within the body of the function. If multiple paths through the code exist, however, the compiler will not verify that an assignment to the function result is on each path.

The function result may be of any scalar, record, array, set, or pointer type but may not be a file type or a structured type having a file component. Although it may appear in expressions, assignment (:=) is the only operation allowed on the function result. You cannot access individual array components or record fields of the function result; nor can you access the storage to which a function result of a pointer type refers. You cannot pass a function identifier to a formal VAR parameter, and you cannot use the function identifier as the parameter to the predeclared ADDRESS function (see Section 8.2).

A subprogram block may refer to a function identifier declared in an enclosing block but only for the purpose of assigning a value to it. If you use the function identifier as an expression within its own executable section, the result is a recursive call to the function rather than a variable reference.

6.4.3 Subprogram Examples

The following examples show complete procedure, function, and process declarations:

1. This procedure computes the minimum, maximum, and average values in array A. Min, Max, and Avg are formal VAR parameters. Their values are returned to the calling block and can be used in further computations in the program. A is specified as a value parameter, because the procedure is concerned only with the values in the array; the array is not an output of the procedure.

```
PROCEDURE Min_Max_Avg
  (VAR A : ARRAY [L..H:INTEGER] OF range;
  VAR Min, Max : Range; VAR Avg : REAL);

  VAR
    Sum, J : INTEGER;

  BEGIN
    Max := A[L];
    Min := Max;
    Sum := Max;
    FOR J := L+1 TO H DO
      BEGIN
        Sum := Sum + A[J];
        IF A[J] > Max
        THEN
          Max := A[J];
        IF A[J] < Min
        THEN
          Min := A[J];
      END;
    Avg := Sum/(H - L+1);
  END;
```

2. This function uses two VAR parameters: String and Key. Count_substr returns an integer value indicating the number of times Key appears within String.

```
FUNCTION Count_substr
(VAR String: [READONLY] PACKED ARRAY [Ls..Us: INTEGER] OF CHAR;
VAR Key: [READONLY] PACKED ARRAY [Lk..Uk: INTEGER] OF CHAR): INTEGER;

(* This function returns the number of times one substring *)
(* is found in another. *)

 LABEL
   10;
 VAR
   I, J, K: INTEGER;
 BEGIN
   K := 0;
   FOR I := Ls TO Us - Uk - Lk + 2 DO
     IF String[I] = Key[Lk]
       THEN
         BEGIN
           FOR J := 1 TO Uk - Lk DO
             IF String[I + J] <> Key[J + 1]
               THEN GOTO 10;
           K := K + 1;
         10: END;
   Count_substr := K;
 END (* Count_substr *) ;
```

3. This example shows the declaration of a process that, when invoked, waits on a queue semaphore until a packet of data becomes available, then copies it into a local buffer.

```
VAR
  Queue_1 : QUEUE_SEMAPHORE_DESC;
  Buffer : PACKED ARRAY [0..511] OF CHAR;
  Length : 0..512;
  Info : INFO_BLOCK;

[PRIORITY(10), STACK_SIZE(100), NAME ('CONSUM')] PROCESS Consumer;
BEGIN
  WHILE TRUE DO
  BEGIN
    (* Receive one full buffer by value. *)
    RECEIVE ( VAL_DATA := Buffer,
              VAL_LENGTH := Length,
              RET_INFO := Info,
              DESC := Queue_1 );
                        .
                        .
                        .
      (* Process the data *)
                        .
                        .
                        .
  END;
END; (* Process Consumer *)
```

6.5 Directives

A directive is the alternative to a block in a subprogram declaration. A directive gives the compiler information about subprograms for which you declare the heading separately from the body, indicated by the FORWARD directive, and subprograms that are external to a Pascal program, indicated by the EXTERNAL and SEQ11 directives.

To specify a directive, include it as the last item in the subprogram heading and follow it with a semicolon (;). Directives are recognized only in that position in the heading. When you use a directive, the heading must not be followed by a block. The following subsections describe the two classes of directives.

6.5.1 FORWARD Directive

Although Pascal requires you to declare subprograms before you refer to them, a forward declaration allows a subprogram to reference another subprogram before its block has been declared. For example, if two subprograms call each other recursively, a complete declaration of both subprograms is impossible. Omitting the declaration is also impossible, because Pascal needs information about a subprogram's formal parameters before it can compile a reference to the subprogram. You must therefore forward-declare one of the recursive subprograms. The forward declaration gives the compiler the information it needs, just as any other declaration does. However, the forward declaration allows you to withhold the declaration of the subprogram block until later in the source file.

A forward declaration consists of the subprogram heading followed by the FORWARD directive, without a subprogram block. For example:

```
PROCEDURE Chestnut
  (Bld : REAL; Doc : CHAR;
   VAR Arc : Rec); FORWARD;
```

This example forward-declares the procedure Chestnut. The forward declaration includes only the information shown in the example.

When you specify the block of a forward-declared subprogram, you supply only the appropriate reserved word (PROCEDURE, FUNCTION, or PROCESS) and the subprogram name. You do not repeat any other information that appears in the subprogram heading.

Example

This example forward-declares the function Adder. The function block appears after the declaration of the procedure Printer. The heading of the Adder block describes its formal parameters and result type within comment delimiters. Although you must omit the parameter list and result type when you declare the function block, inserting them as a comment is good documentation practice.

```
FUNCTION Adder (Op1, Op2, Op3 : REAL) : REAL; FORWARD;
PROCEDURE Printer (Student : Name_Array);
  .
  .
  .
  BEGIN
    .
    .
    Z := Adder (A, B, C);
    .
  END;
FUNCTION Adder (* (Op1, Op2, Op3 : REAL) : REAL *) ;
  .
  .
  .
  BEGIN
    .
    .
    .
    Printer ('Leonardo da Vinci');
    .
    .
    .
  END;
```

6.5.2 External Subprograms

The EXTERNAL and SEQ11 directives let you declare Pascal and MACRO–11 subprograms that are external to a main program. Those directives specify that a procedure, function, or process resides in an independently compiled module (Chapter 7). The subprogram declaration within the other module must include the GLOBAL attribute (see Section 10.2.9).

6.5.2.1 EXTERNAL Directive

The EXTERNAL directive causes the compiler to generate a standard MicroPower/Pascal calling sequence. Use this directive whenever you want to specify an external subprogram that you have written either in Pascal or in MACRO–11.

The name that you use in the subprogram heading must be unique; that is, no two global subprograms can have the same name. You may use the EXTERNAL and GLOBAL attributes to create a global name for a subprogram that is different from the subprogram name.

MACRO–11 subprograms must be explicitly coded to respond to this calling sequence so calls and parameters are passed correctly. Appendix B of the *MicroPower/Pascal Run-Time Services Manual* provides complete specifications for this calling sequence.

Examples

1. This example declares MTH as an external routine.

   ```
   FUNCTION MTH (Angle : REAL) : REAL; EXTERNAL;
   ```

2. This example declares the process Alarm. The attributes specify that the default run-time name for the process is Alarm1 and that its priority is 20. The formal parameter list specifies I as an integer parameter with a default value of 52.

   ```
   [NAME('Alarm1'), PRIORITY(20)] PROCESS  Alarm
   (I:INTEGER:= 52); EXTERNAL;
   ```

6.5.2.2 SEQ11 Directive

The SEQ11 directive causes the compiler to generate a standard PDP–11 FORTRAN calling sequence. Use that directive whenever you want to access MACRO–11 assembly language routines that were written to be called from a FORTRAN program. Appendix B of the *MicroPower/Pascal Run-Time Services Manual* provides complete specifications for this calling sequence. That directive is illegal in PROCESS declarations.

Note

Routine declarations using that directive must specify all parameters as variable (VAR) parameters to ensure that they are passed correctly. Failure to comply with this rule will not be detected by the compiler.

Example

This example declares the FORTRAN library procedure $FSTRG. The formal parameter list specifies S as a parameter.

```
[EXTERNAL($FSTRG)] PROCEDURE Forstring (VAR S : INTEGER); SEQ11;
```

6.6 Activating Procedures, Functions, and Processes

A Pascal subprogram executes in response to an activation request in the executable section of a program or subprogram. Those requests are procedure calls, function identifiers, and process invocations.

The syntax for activating procedures, functions, and processes is the same, but the ways in which you use procedure calls, function identifiers, and process invocations within an executable section are different. A procedure call and a process invocation are statements by themselves. A function identifier cannot appear by itself; it is an expression whose resulting value is used within an executable statement.

For example, you could invoke the procedure Yearly_Totals as follows:

```
Yearly_Totals (Amount_Purchased, Amount_Sold, Amount_Discount);
```

You might invoke the function Compute_Interest like this:

```
Earnings := Compute_Interest (Investment, 0.13, 5);
```

The procedure Yearly_Totals is executed for its effects; the function Compute_Interest is executed to compute a value that is then assigned to the variable Earnings.

A process invocation might appear in your program as:

```
Move_Data (in, out, NAME:='data01', PRIORITY:= 30,
                    DESC:=mdesc, STACK_SIZE:= 3000 )
```

The procedure call and process invocation are described in Chapter 5, along with the other executable Pascal statements.

The MicroPower/Pascal language allows much flexibility in specifying the association of formal and actual parameters. You can call a routine or invoke a process at different times with different actual parameters. Whether an actual parameter is legal depends on the kind of formal parameter to which it is being passed.

6.6.1 Parameter Association

The parameter list passed during subprogram activation must include exactly one actual parameter for each formal parameter. The actual parameter is either specified explicitly in the activating statement or supplied as a default value in the formal parameter list of the subprogram declaration. You may establish the correspondence between actual parameters mentioned in the call and formal parameters by using either positional or nonpositional syntax.

Positional syntax associates actual parameters with formal parameters solely on the basis of position in the respective parameter lists. You must specify the actual parameters in the positional order established in the formal parameter declaration. That is, the association of actual and formal parameters proceeds from left to right, item by item, through both lists.

Nonpositional syntax associates the actual parameters with formal parameters, without regard to their position in the parameter lists, by specifying both the formal parameter keyword and the actual parameter in an assignment (:=) statement. Because the association is by name, the parameters in the call do not have to appear in the same order as the formal parameters appeared in the declaration.

Nonpositional Syntax

{ formal-parameter := actual-parameter } ,...

formal-parameter

The parameter name specified in the formal parameter list of the subprogram declaration.

actual-parameter

The actual parameter to be associated with the formal parameter.

You may include both positional and nonpositional actual parameters in the same subprogram activation statement. You must still, however, supply one actual parameter for each formal parameter, whether it be a positional, a nonpositional, or a default parameter. If you use both positional and nonpositional parameters, you must list the positional parameters first, separated by commas.

Examples

1. Suppose that you have declared the following procedure:

   ```
   PROCEDURE Compute_Sum  (X, Y : INTEGER; VAR Z : INTEGER);
   ```

   Using positional syntax, you could issue the following procedure call:

   ```
   Compute_Sum (Quantity + 6, 15, Total);
   ```

   Formal parameter X is thus passed the value of Quantity + 6; Y is passed the integer value 15; and Z is passed the variable Total.

2. Using nonpositional syntax, you could call the procedure Compute_Sum with the following statement:

   ```
   Compute_Sum (Z := Total, X := Quantity + 6, Y := 15);
   ```

   This call to Compute_Sum is equivalent to the call above that used positional syntax.

3. If you used both positional and nonpositional actual parameters in the same parameter list, the call above to Compute_Sum might look like this:

   ```
   Compute_Sum (Quantity + 6, Z := Total, Y := 15);
   ```

   The first actual parameter, Quantity + 6, corresponds to the formal parameter X because they are the first parameters in their respective lists. Since the next two actual parameters use nonpositional syntax, you must specify the formal parameters to which they belong.

6.6.2 Default Parameters

When a call to a subprogram supplies no actual parameter for a formal parameter that was declared with a default value, the default is used. You declare a default parameter in the formal parameter list of a subprogram (see Sections 6.3.2 and 6.3.3).

You may use positional or nonpositional syntax (see Section 6.6.1) when selecting a default parameter. If you use positional syntax, omit the actual parameter and indicate the position of its default with a comma. If you use nonpositional syntax, omit the parameter from the list. For example, suppose that you declare the following function:

```
FUNCTION Net_Pay
  (Hours : INTEGER; Tax : REAL := 0.05;
   Rate : REAL; Fica : REAL := 0.07;
   Overtime : INTEGER) : REAL;
```

The formal parameters Tax and Fica are given default values of 0.05 and 0.07, respectively. You need to pass actual parameters only to the formal parameters Hours, Rate, and Overtime. You may call the function Net_Pay in several ways, as illustrated in the following example:

```
Take_Home_Year := Take_Home_Year +
   Net_Pay (Overtime := Overtime_Week, Rate := Pay_Rate, Hours := Hours_Week);

Take_Home_Year := Take_Home_Year +
   Net_Pay (Hours_Week, ,Pay_Rate, , Overtime_Week);
```

You can override the default values of a formal parameter by associating it with an actual parameter. For example, if you wanted to replace the default value of the formal parameter Tax in the example above, you could call Net_Pay as follows:

```
Take_Home_Year:= Take_Home_Year +
   Net_Pay (Hours_Week, 0.06, Pay_Rate, , Overtime_Week);
```

As a result, the default value of Tax would be replaced by the value 0.06, supplied in the actual parameter list.

6.6.3 Actual Value Parameters

When a subprogram requires an actual parameter solely for an input value, you may use value semantics to pass the actual parameter. An actual value parameter must be an expression that is assignment-compatible with the formal value parameter to which it corresponds. Because there is no assignment compatibility for file variables, they can never be passed as value parameters.

If necessary, the type of an actual parameter is converted to the type of the formal parameter to which it is being passed. Pascal does so by following the same type-conversion rules it uses to perform any other assignment. You may, for example, pass an integer expression to a formal parameter of a real type. If the formal parameter has the UNSAFE attribute, no conversion occurs (see Section 10.2.26).

The following formal parameter list requires three value parameters:

```
PROCEDURE Alpha (A, B : INTEGER; C : CHAR);
```

You could write the following procedure call, with X and Y as integer variables, for the procedure Alpha:

```
Alpha (X+Y, 11, 'G');
```

The actual parameters corresponding to A and B must be integer expressions and the actual parameter corresponding to C must be a character expression.

6.6.4 Actual VAR Parameters

When a subprogram requires an actual parameter as output, you must use VAR semantics to pass the actual parameter. Because the subprogram has direct access to the variable, any change the subprogram makes to its value is reflected in the actual parameter when control returns to the calling block. An actual parameter corresponding to a formal VAR parameter must be a variable in an unpacked context; an actual parameter cannot be a constant unless the formal VAR parameter has the READONLY attribute (see Section 10.2.21). You must pass file variables as VAR parameters.

Caution

If a process invoked within a routine is passed a local (nonstatic) variable to a formal VAR parameter, you must ensure that the routine does not exit before the process terminates. Otherwise, any variable data to be passed to that process using formal VAR parameters must be static to avoid erroneous results. Statically allocated variables are created either when you declare them at the outermost level of a program or when you use the AT, EXTERNAL, GLOBAL, or STATIC attributes.

When passing arrays and character strings to formal conformant array parameters, you must make sure that the components and indexes of both parameters are of the same base type.

Before passing a character string constant to a formal packed conformant array parameter, ensure that the formal parameter was declared READONLY (see Sections 6.3.3.1 and 10.2.21).

The index bounds of an actual array parameter must be within the bounds specified by the conformant array in the formal parameter declaration.

A variable passed to a subprogram as an actual VAR parameter must have the identical type of the corresponding formal parameter. You cannot pass a component of a packed structure to a formal VAR parameter, although you can pass the entire structure.

Certain attributes of subprogram parameters affect the rules of compatibility between actual and formal VAR parameters. The resulting modifications to structural compatibility rules are outlined below. Those rules also apply to the corresponding components of structured types and pointer types used as formal parameters.

- Volatility—A VOLATILE actual VAR parameter may not be passed to a formal VAR parameter that is not VOLATILE.
- Accessibility—A READONLY actual VAR parameter can be passed only to a READONLY formal VAR parameter. Likewise, a WRITEONLY actual VAR parameter can be passed only to a WRITEONLY formal VAR parameter.
- Unsafe—An UNSAFE formal VAR parameter will accept an actual parameter if the size of the actual parameter is greater than or equal to the size of the formal parameter. If the formal parameter is declared with the READONLY attribute, this size restriction does not apply.

The following formal parameter list contains three VAR parameters:

```
PROCEDURE Tempest (VAR Sea, Breeze : REAL; VAR Sick : Med_File);
```

You could call the procedure Tempest with this statement:

```
Tempest (Tide, Speed, Patient);
```

The actual parameters Tide and Speed must be variables of type REAL. The actual parameter Patient must be a variable of the previously defined type Med_File.

6.6.5 Actual Procedure and Function Parameters

Sometimes a subprogram requires the name of a procedure or function as an actual parameter. When passing routines to other subprograms, you must make sure that the formal parameter lists in both declarations are congruent. As described in Section 6.3, a formal parameter list can have value, VAR, procedure, and function parameter sections. Two formal parameter lists are congruent if they have the same number of sections and if the sections in corresponding positions meet any of the following conditions:

- They are both value parameter sections containing the same number of parameters of compatible types (see Section 2.9.2).
- They are both VAR sections containing the same number of parameters. The parameters must either be of identical types or be equivalent conformant arrays. Any attributes associated with a formal VAR parameter affect the kinds of actual parameters that can be passed (see Section 6.6.4).
- They are both procedure identifier sections having congruent formal parameter lists or no formal parameters at all.
- They are both function identifier sections with congruent formal parameter lists or no formal parameters at all and with compatible result types.

Chapter 7
Compilation Units

The MicroPower/Pascal software permits as compilation units programs and modules. Although the structures of the two are similar, programs have executable blocks at the outermost level, whereas modules do not. A program can be compiled, built, and executed by itself with only the various system modules that are included automatically. A module, on the other hand, cannot be executed unless it is merged with a main program written in Pascal or MACRO–11. Pascal gives you the option of writing modules that can be:

- Combined with a program and other separately compiled but logically coordinated modules for execution as a single unit
- Developed independently from other programs or modules but used as library modules bound into larger entities at build time

The %INCLUDE directive simplifies program coding by allowing commonly used declarations and statements to reside in a single file. Each compilation unit that uses these declarations and statements references the file in a %INCLUDE directive where needed (see Section 7.3).

7.1 Compilation Unit Structure

A MicroPower/Pascal compilation unit begins with a heading that identifies it as either a program or a module.

Syntax

```
⟦ [ {attribute} ,... ] ⟧ { PROGRAM } identifier ⟦ ( {file-variable} ,... ) ⟧ ;
                          { MODULE  }
⟦ { label-declaration       }                                 ⟧
⟦ { constant-declaration    }                                 ⟧
⟦ { type-definition         } ... BEGIN ⟦ {statement} ;... ⟧  ⟧ END .
⟦ { variable-declaration    }                                 ⟧
⟦ { subprogram-declaration  }                                 ⟧
```

attribute

Additional information about the compilation unit, provided through the attributes: CONTEXT, DATA_SPACE, DEV_ACCESS, DRIVER, GROUP, INIT_PRIORITY, OVERLAID, PRIORITY, PRIVILEGED, STACK_SIZE, and SYSTEM (see Chapter 10).

PROGRAM

The statements that make up a declaration section and an executable section to form a main program.

MODULE

The compilation unit. A module may contain declarations, including subprograms. Any executable statement within the module must be contained within a subprogram declaration.

identifier

The name of the program or module.

file-variable

The external file variables used by the program or module. MicroPower/Pascal does not interpret the file variable list, and you need provide it only to be consistent with the requirements of standard Pascal. In particular, MicroPower/Pascal assumes the default program parameter declarations of INPUT and OUTPUT (see Sections 2.7 and 9.2.9). MicroPower/Pascal also differs from standard Pascal by allowing you to redefine the INPUT and OUTPUT identifiers at the program level.

label-declaration

A label declaration, as defined in Section 4.2.

constant-declaration

A constant declaration, as defined in Section 4.1.

type-definition

A type definition, as defined in Section 4.3.

variable-declaration

A variable declaration, as defined in Section 4.4.

subprogram-declaration

A subprogram declaration, as defined in Section 6.2.

statement

One of the MicroPower/Pascal statements, described in Chapter 5.

Restriction

Within a declaration section, CONST, LABEL, TYPE, and VAR sections should appear first, followed by any subprogram declarations. However, subprogram declarations with the EXTERNAL directive or the EXTERNAL attribute may be placed anywhere in the declaration section.

7.3 The %INCLUDE Directive

The %INCLUDE directive inserts source text from one source file into another source file during compilation. The contents of the included file are inserted at the point where the MicroPower/Pascal compiler encounters the %INCLUDE directive. That command can appear anywhere that a comment is legal and is often used when the same information resides in several compilation units. Appendix H discusses compiler limitations and the use of type definitions in %INCLUDE files.

Syntax

%INCLUDE 'file-specification'

file-specification

The file to be included. You must enclose the file specification within apostrophes. Refer to the applicable MicroPower/Pascal system user's guide for the file-specification syntax.

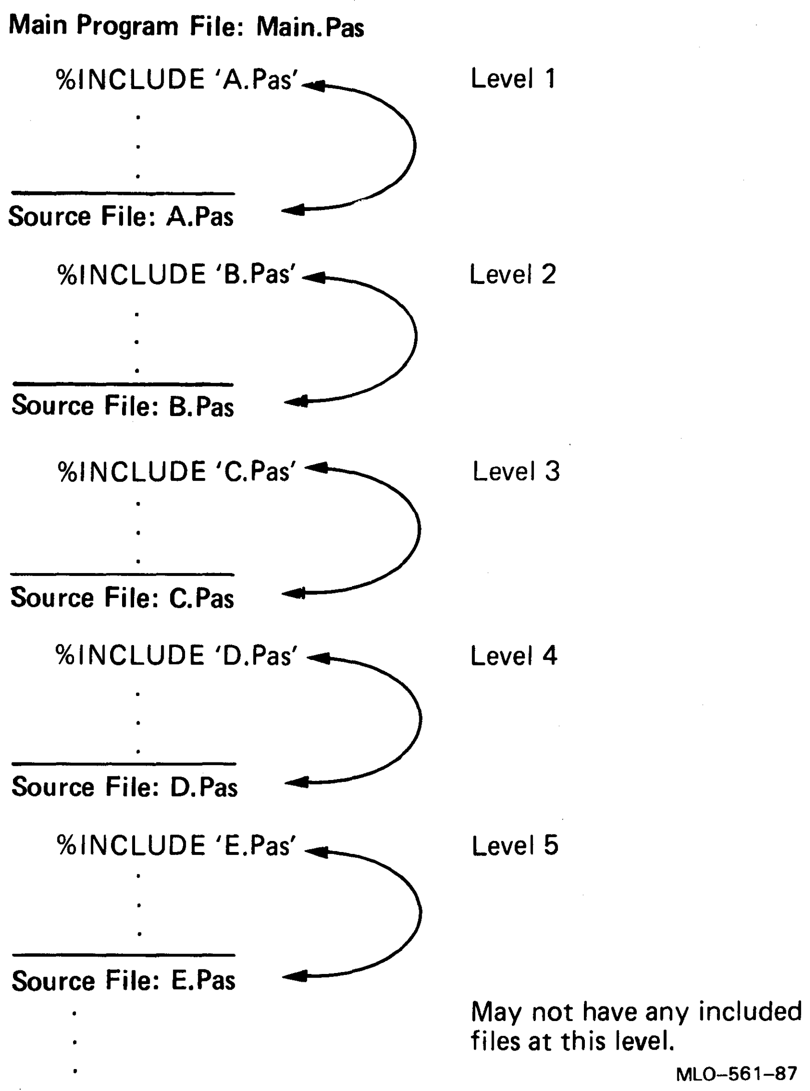
Rules and Defaults

- You must refrain from using the %INCLUDE directive recursively. The compiler does not detect this error condition.
- You may not place comment text between the %INCLUDE keyword and its file-specification.

When the compiler encounters the %INCLUDE directive, the compiler saves its position in this file and begins reading from the included file. Upon reaching the end of the included file, the compiler resumes reading the original file at the point immediately following the %INCLUDE directive.

An included file at the outermost level of a program is said to be included at the first level. A file included by a first-level include file is at the second level, and so on. A program may not include any files beyond the fifth level. Figure 7–1 illustrates the legal levels of included files.

Figure 7-1: %INCLUDE File Levels



Examples

1. The following %INCLUDE directive specifies the file CONDEF.PAS, which contains constant definitions.

```
(* File CONDEF.PAS *)

     Max_Class = 300;
     N_Profs = 140;
     Frosh = 3000;

(* Main Pascal Program *)

[SYSTEM(MICROPOWER)] PROGRAM Student_Courses;
   CONST %INCLUDE 'CONDEF.PAS'
   TYPE
     Schedules = RECORD
                    Year : (FR, SO, JR, SR);
                    Name : PACKED ARRAY [1..30] OF CHAR;
                    Parents : PACKED ARRAY [1..40] OF CHAR;
                    College : (Arts, Engineering, Architecture, Agriculture, Hotel)
                    END;
                      .
                      .
                      .
```

   The %INCLUDE directive instructs the compiler to insert the contents of the file CONDEF.PAS after the reserved word CONST in the main program. The main program Student_Courses is compiled as if it contained the following:

```
[SYSTEM(MICROPOWER)] PROGRAM Student_Courses;
   CONST
     Max_Class = 300;
     N_Profs = 140;
     Frosh = 3000;
   TYPE
     Schedules = RECORD
                    Year: (Fr, So, Jr, Sr);
                    Name: PACKED ARRAY [1..30] OF CHAR;
                    Parents : PACKED ARRAY [1..40] OF CHAR;
                    College : (Arts, Engineering, Architecture, Agriculture, Hotel)
                    END;
                      .
                      .
                      .
```

2. Although you may use %INCLUDE directives in another included file, you should not use them recursively. If, for example, the file OUT.PAS contains a %INCLUDE directive for the file IN.PAS, IN.PAS should not contain the command %INCLUDE for the file OUT.PAS.

Chapter 8
Utility Routines

The MicroPower/Pascal software supplies predeclared procedures and functions that perform commonly required operations. Predeclared functions always return a value that is associated with the function name. Table 8–1 lists those routines by functional category.

Table 8–1: Utility Routines by Functional Category

Name	Description
	Arithmetic Functions
ABS	Computes absolute value
ARCTAN	Computes arctangent
COS	Computes cosine
EXP	Computes exponential
LN	Computes natural logarithm
SIN	Computes sine
SQR	Computes square
SQRT	Computes square root
	Ordinal Functions
PRED	Returns value that precedes x
SUCC	Returns value that follows x
	Boolean Functions
ODD	Determines whether a value is odd or even

Table 8–1 (Cont.): Utility Routines by Functional Category

Name	Description
	Transfer Routines
CHR	Converts integers to characters
LROUND	Converts a real value to a long integer by rounding the fraction
LTRUNC	Converts a real value to a long integer by truncating the fraction
ORD	Obtains ordinal position of a member of an ordinal type
PACK	Converts an unpacked array to a packed array
ROUND	Converts a real value to an integer by rounding the fraction
SHORT	Converts a long integer value to an integer by truncating the most significant 16 bits of the value
TRUNC	Converts a real value to an integer by truncating the fraction
UNPACK	Converts a packed array to an unpacked array
UROUND	Converts a real value to an unsigned value by rounding the fraction
USHORT	Converts a long integer value to an unsigned value by truncating the most significant 16 bits of the value
UTRUNC	Converts a real value to an unsigned value by truncating the fraction
	Pointer Routines
ADDRESS	Obtains a pointer to a variable
DISPOSE	Deallocates memory used by a dynamic variable
NEW	Allocates memory to contain a dynamic variable
	Unsigned Functions
UAND	Performs bit-by-bit AND of its parameters
UNOT	Performs bit-by-bit one's complement of its parameter
UOR	Performs bit-by-bit OR of its parameters
UXOR	Performs bit-by-bit EXCLUSIVE–OR of its parameters

Table 8–1 (Cont.): Utility Routines by Functional Category

Name	Description
	Allocation Size Functions
BITNEXT	Obtains size, in bits, of a data item in a packed array
BITSIZE	Obtains size, in bits, of a packed record field
NEXT	Obtains size, in bytes, of a data item in an unpacked array
SIZE	Obtains size, in bytes, of a data item

The following sections describe the predeclared MicroPower/Pascal utility routines in alphabetical order by name.

8.1 ABS(x) Function

The ABS(x) function computes the absolute value of a number. The parameter (x) is an expression of type INTEGER, LONG_INTEGER, UNSIGNED, or REAL. The function returns a result type that is the same type as the parameter.

Note

The function does not detect the presence of the overflow condition that occurs when the parameter (x) has the integer value –32,768.

8.2 ADDRESS(x) Function

The ADDRESS(x) function returns a pointer value that references the parameter (x). The parameter (x) is any data item that is the identifier of a variable or a formal parameter.

8.3 ARCTAN(x) Function

The ARCTAN(x) function computes the arctangent. The parameter (x) is an expression of type INTEGER, LONG_INTEGER, UNSIGNED, or REAL. The function result is a value of type REAL expressed in radians.

8.4 BITNEXT Function

The BITNEXT function determines the number of bits allocated for a single component of the specified type in a packed array. The function result is an INTEGER value.

Syntax

BITNEXT (x ⟦ {,t} ... ⟧)

x

A type identifier or a variable; treated as if it were a component of that type in a packed array.

t

An ordinal constant that represents a nested tag field value. The first constant of a series of constants represents the outermost variant.

If parameter (x) is a variant, you may supply one or more additional parameters (t): the case constant corresponding to a variant of the record. See Appendix E for the default allocation sizes of standard data types.

Note

The BITNEXT and BITSIZE functions return the same bit size values for a particular type, unless the components of the specified type in a packed array would have been padded to ensure proper alignment.

8.5 BITSIZE Function

The BITSIZE function determines the number of bits allocated for a packed record field of that type. The function returns a value of type INTEGER.

Syntax

BITSIZE (x ⟦{,t} ... ⟧)

x

A type identifier or a variable; treated as if it were a packed record field of that type.

t

An ordinal constant that represents a nested tag field value. The first constant of a series of constants represents the outermost variant.

If parameter (x) is a variant, you may supply one or more additional parameters (t): the case constant corresponding to a variant of the record. See Appendix E for the default allocation sizes of standard data types.

Note

The BITSIZE and BITNEXT functions return the same bit size values for a particular type, unless the components of the specified type in a packed array would have been padded to ensure proper alignment.

8.6 CHR(x) Function

The CHR(x) function returns a value of type CHAR whose ordinal position in the character set is specified by the parameter. The parameter (x) is an integer value from 0 to 255.

8.7 COS(x) Function

The COS(x) function computes the cosine of an angle. The parameter (x) is an expression of type INTEGER, LONG_INTEGER, UNSIGNED, or REAL that is the angle expressed in radians. The function result is a value of type REAL.

8.8 DISPOSE(p) Procedure

The DISPOSE(p) procedure deallocates memory occupied by a dynamic variable and returns the space to the heap. The parameter (p) is a pointer variable.

Examples

1. This example deallocates memory for the dynamic variable Ptr.

   DISPOSE(Ptr);

   As a result, the memory allocated for Ptr^ is deallocated, and the variable is destroyed. The value of Ptr is now undefined.

2. This program constructs a linked list of records. Each student record contains data on one student (a name and a student ID number) and a field that is a pointer to the next record. The program reads a number and a name and assigns each of them to a field of the student record. Then the program inserts the new component at the beginning of the linked list by assigning the "Start" pointer to that new record.

```
[SYSTEM(MICROPOWER)]PROGRAM LinkedList;
   TYPE
     Student_Ptr = ^Student_Data;
     String = PACKED ARRAY[1..20] OF CHAR;
     Number = 1..9999;
     Student_Data = RECORD
                     Name : String;
                     Stud_ID : Number;
                     Next : Student_Ptr;
                   END;
     VAR
       Start, Student : Student_Ptr;
       New_ID : Number;
       New_Name : String;
       Count : INTEGER;

     PROCEDURE Write_Data(Student : Student_Ptr);

(* This procedure prints the list of students.  Because *)
(* the printing starts at the beginning of the linked *)
(* list, the student names and ID numbers are printed in *)
(* the reverse of the order in which they were entered. *)
```

```
VAR
  I,J : INTEGER;
  Next_Student : Student_Ptr;
BEGIN
  WRITELN ('Name:', 'Student ID#:':29);
  REPEAT
  WRITELN(Student^.Name : 20, Student^.Stud_ID : 7);
  Next_Student := Student^.Next;
  DISPOSE (Student);
  Student := Next_Student
  UNTIL Student = NIL;
END; (* End of Write_Data *)

(* Main Program *)
BEGIN
  Count := 0;
  WRITELN ('Type a 5-digit ID number and a name for each student.');
  WRITELN('Press CTRL/Z when finished.');
  Start := NIL;
  WHILE NOT EOF DO
    BEGIN
      READLN (New_Id, New_Name);
      NEW (Student);
      Student^.Next := Start;
      Student^.Name := New_Name;
      Student^.Stud_Id := New_Id;
      Start := Student;
      Count := Count + 1;
    END;
  IF Count > 0
  THEN
    Write_Data(Start);
END.
```

In the main program, the WHILE loop reads a number and a name for one student. The following procedure call allocates memory for a new student record:

```
NEW(Student);
```

The new record is inserted at the beginning of the list: at Student^. "Next" points to the previous head of the list. The value of the new student record is assigned to the Start pointer.

The Write_Data procedure writes the name and student ID number for each student in the linked list. After writing data for one student, the procedure assigns the address of the next record in the list to Next_Student. The following call deallocates memory for one student record:

```
DISPOSE(Student);
```

After deallocating memory, the procedure assigns the value of Next_Student to Student. When the current Student record points to NIL, the loop stops executing.

8.9 DISPOSE Procedure: Record-with-Variants Form

Use this form of the DISPOSE procedure when manipulating dynamic variables of a record type with variants.

Syntax

DISPOSE (p {,t} ...)

p

A pointer value with a type that points to a record with variants.

t

An ordinal constant that represents a nested tag field value. The first constant of a series of constants is the outermost variant.

This form of DISPOSE releases memory occupied by the variable referenced by p. The tag field values should be identical to those specified when memory was allocated with the NEW procedure (see Section 8.14).

Example

```
DISPOSE(Menu_Selection, Beef, Oz_32);
```

This call deallocates the memory allocated by the last NEW procedure call. If a dynamic variable with specified record variants was allocated by the NEW procedure, the variable should be deallocated only by the DISPOSE procedure specifying identical record variants.

8.10 EXP(x) Function

The EXP(x) function computes the exponential of its parameter. The parameter (x) is an expression of type INTEGER, LONG_INTEGER, UNSIGNED, or REAL. The function result is a value of type REAL.

8.11 LN(x) Function

The LN(x) function computes the natural logarithm of a number. The parameter (x) is an expression having a value greater than 0 and must be of type INTEGER, UNSIGNED, or REAL. The function result is a value of type REAL.

8.12 LROUND(r) Function

The LROUND(r) function converts a value of type REAL to its representation as a long integer by rounding any fractional part. The parameter (r) is a value of type REAL. Rounding proceeds as follows:

- When the value (r) is positive and the fractional part of the number is 0.5 or greater, 1 is added to the integer part.
- When the value is negative and the absolute value of the fractional part of the number is 0.5 or greater, –1 is added to the integer part.

- When the absolute value of the fractional part of the number is less than 0.5, the integer part is unaffected.

LROUND(r) returns a value that is of type LONG_INTEGER. An exception occurs if the value is too large to be represented by a long integer value.

8.13 LTRUNC(r) Function

The LTRUNC(r) function converts a value of type REAL to its representation as a long integer by truncating any fractional part. The parameter (r) is a value of type REAL. The value returned is of type LONG_INTEGER. An exception occurs if the value is too large to be represented by a long integer value.

8.14 NEW(p) Procedure

The NEW(p) procedure allocates memory for a dynamic variable. The procedure sets aside memory for p^—the variable to which parameter (p) refers. To access the allocated variable, you must dereference the pointer variable by appending a circumflex (^) to the variable's identifier. The value of the newly allocated variable (p^) is undefined; you cannot assume that it contains any meaningful data.

Examples

1. This example declares Ptr as a pointer to an integer variable. However, the integer variable and its address do not yet exist.

   ```
   VAR Ptr: ^Integer;
   ```

2. You use the following procedure call to allocate memory for the dynamic variable:

   ```
   NEW(Ptr);
   ```

   This call allocates a variable of type INTEGER in dynamically allocated heap storage. The variable is denoted by Ptr^: the pointer variable's name followed by a circumflex (^). This call also assigns the address of the allocated integer to Ptr.

8.15 NEW Procedure: Record-with-Variants Form

Use this form of the NEW procedure when manipulating dynamic variables of a record type with variants.

Syntax

NEW (p {,t} ...)

p

A pointer variable with a type that points to a record with variants.

t

An ordinal constant that represents a nested tag field value. The first constant of a series of constants is the outermost variant.

If you create a pointer without specifying the tag field values, the system allocates enough memory to hold any of the variants in the record. Sometimes, however, a dynamic variable will take values of only a particular variant. If that variant requires less memory than NEW(p) would allocate, you can use the form shown above. Because the record-with-variants form of the NEW procedure allocates memory for the specified variant and not for the largest variant in the declaration, you should not assign or evaluate the entire record. You should assign and evaluate only the individual record fields.

Example

```
TYPE
  Menu_Ptr = ^Menu_Order;
  Meat_Type = (Fish, Fowl, Beef);
  Beef_Portion = (Oz_10, Oz_16, Oz_32);
  Menu_Order = RECORD
               CASE Entree : Meat_Type OF
                 Fish : (Fish_Type : (Salmon, Cod, Perch, Trout);
                      Lemon : BOOLEAN);
                 Fowl : (Fowl_Type : (Chicken, Duck, Goose);
                      Sauce : (Orange, Cherry, Raisin));
                 Beef : (Beef_Type : (Steak, Roast, Prime_rib);
                      CASE Size : Beef_Portion OF
                        Oz_10, Oz_16 : (Beef_veg : (Pea, Mixed));
                        Oz_32 : (Stomach_Cure : (Bicarbonate,
                                 Antacid, None_Needed)));
               END;
VAR Menu_Selection : Menu_Ptr;
```

You can allocate memory for only the Fish variant as follows:

```
NEW(Menu_Selection, Fish);
```

The following example shows how to call NEW and to specify tag field values for nested variants:

```
NEW(Menu_Selection, Beef, Oz_32);
```

The tag field values must be listed in the order in which they were declared.

8.16 NEXT(x) Function

The NEXT(x) function determines the number of bytes allocated for a single component of the specified type in an unpacked array. The function result is an INTEGER value.

Syntax

NEXT (x [[{,t} ...]])

x

A type identifier or a variable; treated as if it were a component of that type in an unpacked array.

t

An ordinal constant that represents a nested tag field value. The first constant of a series of constants represents the outermost variant.

If parameter (x) is a variant, you may supply one or more additional parameters (t): the case constant corresponding to a variant of the record. See Appendix E for the default allocation sizes of standard data types.

Note

The NEXT and SIZE functions return the same byte size values for a particular type, unless the components of the specified type in an unpacked array would have been padded to ensure proper alignment.

8.17 ODD(x) Function

The ODD(x) function tests whether a value is odd. The parameter (x) is a value of type INTEGER or UNSIGNED. The function returns TRUE if the value of x is odd and FALSE if the value of x is even.

8.18 ORD(x) Function

The ORD(x) function returns an integer that is the position of x in the ordered sequence of values of that type. The parameter (x) is a value of any ordinal type. The ordinal value of an integer is the integer itself.

8.19 PACK Procedure

The PACK procedure copies the elements of an unpacked array into a packed array.

Syntax

PACK (a, i, z)

a

The identifier of the unpacked array. The component type of this array must be the same as that of array z.

i

The index value in array a where the operation is to begin. This value must be an expression that is assignment compatible with the index type of array a.

z

The identifier of the array to which the elements of array a are copied. This identifier must be a packed array of the same component type as array a.

PACK assigns components of a, starting with a[i], to array z, starting with the first element of z, until all the components in z are filled.

The upper bound of array a (that is, n) must be greater than or equal to i+v–u, where v is the upper bound of array z and u is the lower bound of array z. In other words, ORD(n) > = ORD(i) + ORD(v) –ORD(u).

'The operation of PACK is equivalent to the following:

```
TYPE
  s1 = x..y;
  s2 = u..v;
  rec = RECORD
          .
          .
          .
        END;
VAR
  j : s2;
  k : s1;
  i : s1;
  a : ARRAY [s1] OF REC;
  z : PACKED ARRAY [s2] OF REC;
BEGIN
  k := i;
  FOR j := u TO v DO
    BEGIN
    z[j] := a[k];
    IF j <> v THEN k := SUCC(k)
    END
END
```

Examples

1. This program fragment assigns the components A[1] through A[20] to P[1] through P[20]; that is, all the components in A are packed into P.

```
TYPE
  Somenums = 0..15;
VAR
  A : ARRAY[1..20] OF Somenums;
  P : PACKED ARRAY[1..20] OF Somenums;
  I : INTEGER;
BEGIN
  FOR I := 1 TO 20 DO
    READ (A[I]);
PACK (A,1,P);
END
```

2. The call to PACK in this example moves components of array Data into the packed array Pdata. The index parameter 3 specifies that the packing will start with array component Data[3]. Thus, the 20 components Data[3] through Data[22] are assigned to Pdata[1] through Pdata[20]. The remaining components of the source array, Data[23] through Data[25], will be ignored.

```
TYPE
  Int_15 = 1..15;
VAR
  Data : ARRAY[1..25] OF Int_15;
  PData : PACKED ARRAY[1..20] OF Int_15;
BEGIN
  PACK(Data,3,Pdata);
END
```

8.20 PRED(x) Function

The PRED(x) function returns the value that immediately precedes the value specified by the parameter (x) in the ordered sequence of values associated with that type. The parameter (x) is a value of any ordinal type except LONG_INTEGER. The function result is of the same type as the parameter. An exception condition occurs if x has no predecessor.

8.21 ROUND(r) Function

The ROUND(r) function converts a value of type REAL to its representation as an integer by rounding any fractional part. The parameter (r) is a value of type REAL. Rounding proceeds as follows:

- When the value (r) is positive and the fractional part of the number is 0.5 or greater, 1 is added to the integer part.
- When the value is negative and the absolute value of the fractional part of the number is 0.5 or greater, –1 is added to the integer part.
- When the absolute value of the fractional part of the number is less than 0.5, the integer part is unaffected.

The value returned is of type INTEGER. An exception occurs if the value is too large to be represented by an integer.

8.22 SHORT(l) Function

The SHORT(l) function converts a value of type LONG_INTEGER to its representation as an integer by truncating the most significant 16 bits of the value. An exception occurs if the most significant 16 bits are not 0 or negative. The parameter (l) is a value of type LONG_INTEGER. The value returned is of type INTEGER.

8.23 SIN(x) Function

The SIN(x) function computes the sine. The parameter (x) is an expression of type INTEGER, LONG_INTEGER, UNSIGNED, or REAL that is the angle expressed in radians. The function result is a value of type REAL.

8.24 SIZE Function

The SIZE function determines the number of bytes allocated for a variable or record field of that type. The function returns a value of type INTEGER that indicates the number of bytes allocated by the NEW procedure for a dynamic variable of the specified variant.

Syntax

SIZE (x ⟦{,t} ... ⟧)

x

A type identifier or a variable; treated as if it were a variable or a record field of that type.

t

An ordinal constant that represents a nested tag field value. The first constant of a series of constants represents the outermost variant.

If parameter (x) is a variant, you may supply one or more additional parameters (t): the case constant corresponding to a variant of the record. (Refer to Appendix E for the default allocation sizes of standard data types.)

Note

The NEXT and SIZE functions return the same byte size values for a particular type, unless the components of the specified type in an unpacked array would have been padded to ensure proper alignment.

8.25 SQR(x) Function

The SQR(x) function computes the square of a number. The parameter (x) is an expression of type INTEGER, LONG_INTEGER, UNSIGNED, or REAL. The function returns a result type that is the same type as the parameter.

Note

The function does not detect the presence of an overflow condition.

8.26 SQRT(x) Function

The SQRT(x) function computes the square root of a number. The parameter (x) is an expression of type INTEGER, LONG_INTEGER, UNSIGNED, or REAL. If x has a value less than 0, an exception condition results.

8.27 SUCC(x) Function

The SUCC(x) function returns the value that immediately succeeds the value specified by the parameter (x) in the ordered sequence of values associated with that type. The parameter (x) is a value of any ordinal type except LONG_INTEGER. The function result is of the same type as the parameter.

8.28 TRUNC(r) Function

The TRUNC(r) function converts a value of type REAL to its representation as an integer by truncating any fractional part. The parameter (r) is a value of type REAL. The value returned is of type INTEGER. An exception occurs if the value is too large to be represented by an integer.

8.29 UAND(u1,u2) Function

The UAND(u1,u2) function performs a binary logical AND on the corresponding bits of its two parameters. The parameters (u1,u2) are values of type UNSIGNED. The function result is a value of type UNSIGNED.

Example

```
Result := UAND (%X'751',%X'7A1');
```

The UAND function performs a binary logical AND operation on each pair of bits and returns the unsigned hexadecimal value %X'701'.

8.30 UNOT(u1) Function

The UNOT(u1) function returns the one's complement of its parameter (u1). The parameter (u1) is a value of type UNSIGNED. The function result is a value of type UNSIGNED.

Example

```
Result := UNOT (%B'1000111000111000');
```

The UNOT function performs a binary logical NOT operation on each bit and returns the unsigned value %B'0111000111000111'.

8.31 UNPACK Procedure

The UNPACK procedure copies the elements of a packed array into an unpacked array.

Syntax

UNPACK (z, a, i)

z

The identifier of the array from which elements are being copied. This array must be a packed array of the same component type as array a.

a

The identifier of the array into which elements are being copied. The component type of this array must be the same as that of the array z.

i

The index value in array a where the operation is to begin. This value must be an expression that is assignment compatible with the index type of array a.

UNPACK assigns components of z, starting with z[1], to array a, starting with a[i], until all the components in z are assigned.

The upper bound of array a (that is, n) must be greater than or equal to i+v–u, where v is the upper bound of array z, and u is the lower bound of array z. In other words, ORD(n) > = ORD(i) + ORD(v) –ORD(u).

The operation of UNPACK is equivalent to the following:

```
TYPE
  s1 = x..y;
  s2 = u..v;
  rec = RECORD
          .
          .
          .
        END;
VAR
  j : s2;
  k : s1;
  i : s1;
  a : ARRAY [s1] OF REC;
  z : PACKED ARRAY [s2] OF REC;
BEGIN
  k := i;
  FOR j := u TO v DO
    BEGIN
    a[k] := z[j];
    IF j <> v THEN k := SUCC(k)
    END
END
```

Example

Normally, you cannot pass components of a packed array to a routine using VAR parameters (see Section 6.6.4). If you unpack the array, however, you can pass its components to the routine by reference.

```
VAR
  P : PACKED ARRAY[1..10] OF CHAR;
  A : ARRAY[1..10] OF CHAR;
  I : INTEGER;
PROCEDURE Process_Components (VAR Ch : CHAR);
  BEGIN
    WRITE (''(10,13),Ch);
  END;
BEGIN
  FOR I := 1 TO 10 DO
    BEGIN
      WRITE ('Enter Character');
      READLN (P[I]);
    END
  UNPACK(P,A,1);
  FOR I := 1 TO 10 DO
    Process_Components (A[I]);
END
```

This program reads characters into the packed array P. The procedure call to UNPACK assigns P[1] through P[10] to the unpacked array components A[1] through A[10]. Then, for each call to Process_Components, one component of A is passed to the procedure to print it on the terminal.

8.32 UOR(u1,u2) Function

The UOR(u1,u2) function performs a binary logical OR on the corresponding bits of two parameters. The parameters (u1,u2) are values of type UNSIGNED. The function result is a value of type UNSIGNED.

Example

```
Result  := UOR (%B'10101',%B'10111');
```

The UOR function performs an OR operation on each pair of bits and returns the unsigned value %B'10111'.

8.33 UROUND(r) Function

The UROUND(r) function converts a value of type REAL to its representation as an UNSIGNED type by rounding any fractional part. The parameter (r) is a value of type REAL. Rounding proceeds as follows:

- When the value (r) is positive and the fractional part of the number is 0.5 or greater, 1 is added to the integer part.
- When the value is negative and the absolute value of the fractional part of the number is 0.5 or greater, –1 is added to the integer part.
- When the absolute value of the fractional part of the number is less than 0.5, the integer part is unaffected.

UROUND(r) returns a value that is of type UNSIGNED. An exception occurs if the REAL value is too large to be represented by an unsigned value.

8.34 USHORT(l) Function

The USHORT(l) function converts a value of type LONG_INTEGER to its representation as an unsigned value by truncating the most significant 16 bits of the value. An exception occurs if the most significant 16 bits are not 0 or negative. The parameter (l) is a value of type LONG_INTEGER. The value returned is of type UNSIGNED.

8.35 UTRUNC(r) Function

The UTRUNC(r) function converts a value of type REAL to its representation as an UNSIGNED type by truncating any fractional part. The parameter is a value of type REAL. An exception occurs if the REAL value is too large to be represented by an unsigned value.

8.36 UXOR(u1,u2) Function

The UXOR(u1,u2) function performs a binary logical EXCLUSIVE–OR on the corresponding bits of two parameters. The parameters (u1,u2) are values of type UNSIGNED. The function result is a value of type UNSIGNED.

Example

```
Result := UXOR (%B'0011',%B'0101');
```

The UXOR function performs an EXCLUSIVE–OR operation on each pair of bits and returns the unsigned value %B'0110'.

Chapter 9
Input and Output

This chapter describes the general concepts of I/O processing under MicroPower/Pascal and its predeclared I/O requests. MicroPower/Pascal's extensive set of predeclared routines governing input/output (I/O) processing enable you to establish files on a variety of external I/O resources and to communicate with programs executing on remote computer systems. Table 9–1 lists these predeclared routines.

Table 9–1: Predeclared I/O Routines

Routine	Description
BIN	On output, converts a WRITE or a WRITELN procedure parameter to a binary representation. On input, converts a binary representation of a number in a TEXT file to a READ or READLN procedure parameter.
BREAK	Writes the contents of the current I/O buffer to the I/O server. Operates synchronously (control returns to the caller only when the I/O buffer is empty).
CLOSE	Closes a file.
DELETE_FILE	Deletes a named external file from a directory-structured I/O server.
EMPTY_BUFFER	Initiates an operation to write the contents of the current I/O buffer to the I/O server. When double buffering is selected, operates asynchronously (control returns to the caller immediately after initiating the output operation).
EOF	Tests for the end-of-file condition of an input file.
EOLN	Tests for the end-of-line condition in an input file of type TEXT.
FIND	Moves the file pointer to the specified component.

Table 9–1 (Cont.): Predeclared I/O Routines

Routine	Description
FORMAT_RX02	Formats RX02 flexible diskettes.
GET	Assigns the value of the next component of a file to the buffer variable.
HEX	On output, converts a WRITE or a WRITELN procedure parameter to a hexadecimal representation. On input, converts a hexadecimal representation of a number in a TEXT file to a READ or READLN procedure parameter.
INIT_DIRECTORY	Initializes the directory of a directory-structured I/O server.
OCT	On output, converts a WRITE or a WRITELN procedure parameter to an octal representation. On input, converts an octal representation of a number in a TEXT file to a READ or READLN procedure parameter.
OPEN	Prepares the I/O system to access a specified file and establishes the file's characteristics and access parameters.
PAGE	Sends a form-feed character to an output file of type TEXT.
PROTECT_FILE	Protects an external file on a directory-structured I/O server from deletion.
PURGE	Disconnects a file variable from an I/O server and terminates access to the file.
PUT	Writes the component in a buffer variable to a file.
READ	Reads one or more file components.
READLN	Reads a line of data from a text file.
RENAME_FILE	Renames an external file on a directory-structured I/O server.
RESET	Prepares the file for input and reads the first component.
REWRITE	Prepares a file for output.
SQUEEZE_DIRECTORY	Consolidates the directory entries and all unused blocks on a directory-structured device.
UNPROTECT_FILE	Makes an external file available for deletion.
WRITE	Assigns data to a file.
WRITELN	Writes a line of data to a text file.

9.1 Terminology

The following terms are specific to input and output operations using MicroPower/Pascal.

A Pascal **file** is a collection of logically related components that are arranged in a specific order and treated as a unit.

An **external file** is the physical manifestation of a Pascal file. An external file may be written to or read from direct- or sequential-access storage devices such as A/D converters, disks, logical links, magnetic tapes, and terminals. An external file may be named or unnamed, depending on the device with which the file is associated. The *MicroPower/Pascal I/O Services Manual* describes the physical structure of an external file.

A **named external file** is identified by a name and resides on a directory-structured storage device such as a disk or DECtape II.

An **unnamed external file** resides on a nondirectory-structured device medium. An unnamed file is identified only by the name of the device on which the file resides.

A **directory-structured** device is a direct-access device having a storage medium that contains (at its beginning) a directory of information (file name and length) about all the external files that reside on the medium. Examples of directory-structured devices are disks and DECtape II.

A **nondirectory-structured** device is either a direct- or a sequential-access device having a storage medium that contains no directory of file information. The entire device is treated as a single unnamed external file. Examples of nondirectory-structured devices are A/D converters, DECtape II, disks, logical links, magnetic tape, ring buffers, and terminals.

A **logical link** is a virtual data path connection between two processes. Logical links are created through the network service process (NSP) logical link server (described in the *MicroPower/Pascal I/O Services Manual*).

A **logical link partner** is the process to which a process is connected over a logical link.

A **remote node** is another computer system where the logical link partner resides.

An **I/O server** is a MicroPower/Pascal task that provides an input or output connection between a user's process and an I/O resource. The various MicroPower/Pascal device drivers, the NSP, and the ancillary control process (ACP) are I/O servers.

An **active task** is a process that seeks to establish a connection over a logical link server to a passive task.

A **passive task** is a process that defines itself to a logical link server as being available for connection to an active task.

An **I/O buffer** is the physical buffer between a Pascal buffer variable (Chapter 2) and an I/O server. The size of an I/O buffer is determined by the type of device with which it is associated and can be specified in the OPEN statement.

9.2 I/O Processing

9.2.1 MicroPower/Pascal File Organization

MicroPower/Pascal stores files and maintains directories in the same format as the RT–11 file system. MicroPower/Pascal files are sequentially organized: a file's components are ordered in physical sequence.

Each component, except the first, has another component preceding it and each component, except the last, has another component following it. The physical order in which the components appear is identical to the order in which they were written to the file.

9.2.2 File Access Methods

The access method is the technique a program uses to retrieve and store file components. The access method is specified as part of the OPEN procedure request, which prepares a file for access. A file's access method cannot be changed unless the file is closed (CLOSE procedure) and opened again with a different access method specification. MicroPower/Pascal provides the sequential, direct, and update access methods.

A file may always be processed sequentially, even when the specified access method is direct or update. If the access method is not specified, MicroPower/Pascal defaults to the sequential method.

Sequential access means that file components are processed in the physical sequence in which the components are arranged.

Direct access means that file components are read in an order specified by the FIND request.

Update access means that file components are read in an order specified by the FIND request and may be updated and subsequently written back into the file.

9.2.3 File Variables and I/O Servers

In the MicroPower/Pascal language, as in standard Pascal, you perform I/O operations by using predefined procedures that reference file variables (that is, variables of type FILE) to pass data to and from external files. You can associate those files with a variety of I/O servers that access:

- Nondirectory-structured devices, including A/D converters, DECtape II, disks, logical links, magnetic tape, ring buffers, and terminals.
- Directory-structured devices, including disks and DECtape II.

Note

DECtape II and disks, although traditionally directory structured, may be nondirectory structured as well.

9.2.4 External File Storage

External files, when stored on a magnetic device medium, may be named or unnamed, depending on whether the device medium is to contain one external file or many external files.

You may use magnetic storage devices, such as the RL02 cartridge disk system and the TU58 cartridge/magnetic tape system, to store one or more files on each device medium. When you want to store several files on a device, you create a directory on the device medium that can contain the names of the files residing on the medium. You give an external name to each file that you write on that medium. When the medium is to contain one file only, a directory is not required.

9.2.5 Specifying I/O Servers

The two classes of I/O servers are each specified by a different format. One format applies to the various peripheral devices and ring buffers; the second to using the NSP to establish a logical link connection to another task.

9.2.5.1 Syntax for Specifying External Files and Devices

The following syntax shows how you form an I/O specification for external files, devices, and ring buffers.

The way in which a device or a named external file on a directory-structured device is identified depends on the way in which that device or file will be used. For example:

- To identify a nondirectory-structured device to be treated as a single file or to identify a sequential-access device such as a terminal or a ring buffer, you specify the name of the device only.
- To identify a file on a directory-structured device that contains named external files, you specify the name of the file and the device on which the file resides.

Syntax

device-specification : ⟦filename.type⟧

device-specification

A device name, a ring buffer name, or a logical name. Device names have the form:

ddcuuu

dd

A 2-character device name.

c

A 1-character controller designation character. The default is A.

uuu

A 1- to 3-digit controller unit number. The default is 0.

Ring buffer names and logical names are strings of one to six ASCII characters. Names with less than six characters are automatically padded to six characters with leading spaces. When you substitute a logical name for a device name or ring buffer name, the logical name you choose must translate into a legal 1- to 6-character device name or ring buffer name. See Chapter 20 for information on creating logical names. See the *MicroPower/Pascal I/O Services Manual* for the standard device names.

filename.type

The name and type of the external file. The file name consists of no more than six alphanumeric characters. The type consists of no more than three alphanumeric characters. Uppercase and lowercase forms of a letter are not unique. This parameter is meaningful only when provided with device specifications for directory-structured devices.

Examples

1. This example identifies the file test.dat on drive 0 of RX02 controller A.

```
DYA0:test.dat
```

2. This example identifies the previously created ring buffer IRING.

```
IRING:
```

3. This example identifies drive 0 on the RX02 controller A.

```
DYA0:
```

9.2.5.2 Syntax for Specifying a Logical Link

A logical link specification identifies both active and passive communication tasks to the NSP logical link server.

Note

To completely establish a task as active or passive, the OPEN request must specify HISTORY:= OLD for an active task or HISTORY:= NEW for a passive task.

Syntax

$$\left\{\begin{array}{c} \text{node-address } \llbracket\text{"access-control-string"}\rrbracket\text{::} \\ \text{SY\$NET:} \end{array}\right\} \text{"task-specification-string"}$$

node-address

A unique 2-part number separated by a period that identifies the remote node to be accessed. Specifying a node address declares to the NSP that this task is an active task. The form is:

area . number

area

An integer value in the range 0 to 63. A value of 0, or no value, will cause the default value specified in the NSP prefix file to be used (see the *MicroPower/Pascal I/O Services Manual*).

The period that separates the two parts.

number

An integer value in the range 1 to 1023.

As an alternative to providing a node address, you may substitute a logical name that translates into a string of up to seven ASCII characters, which is the desired node address. See Chapter 20 for information on creating logical names.

Note

The naming convention governing DECnet node names requires that the node name must be a 1- to 6-alphanumeric character name that contains at least one alphabetic character. If your MicroPower/Pascal application communicates with non-MicroPower/Pascal DECnet nodes, you may wish to restrict the logical names that you create to this standard.

access-control-string

A 3-field string, separated by spaces, containing log-in information that is sent to the remote node. This string designates the remote account where tasks reside that will execute in your behalf. The string is in the form:

user-identifier password account

user-identifier

A 1- to 39-character user identifier string.

password

A 1- to 39-character password string.

account

A 1- to 39-character account string.

For more information on access control string formats, see the DECnet documentation applicable to the operating system of the node you wish to access. Nodes operating under MicroPower/Pascal do not require this string.

SY$NET:

Declares to the NSP that the task is passive, that is, available for access by active tasks.

task-specification-string

A string that does the following:

- For an active task, identifies the name or object type of the passive task to which the string is attempting a logical link connection
- For a passive task, declares the name or object type by which an active task can make a logical link connection

Tasks are identified across logical links by a discrete identifier that may be either a name or an object type, as shown below:

$$\left\{ \begin{array}{l} \left\{ \begin{array}{l} \text{TASK =} \\ \text{0 =} \end{array} \right\} \text{task-name} \\ \text{object-type =} \end{array} \right\}$$

task-name
A 1- to 16-character string that uniquely identifies the task. You may optionally substitute 0 in place of the word TASK—"0=MYTASK."

object-type
An integer in the range 1 to 255 that uniquely identifies the task. By convention, object types in the range 1 to 127 are reserved by DIGITAL for generic DECnet system services. DIGITAL suggests that you assign your tasks to object types in the range 128 to 255, unless they are to communicate with those system services.

Examples

1. The following statements establish a task as available for task-to-task communication. The first statement identifies the task as GOOBAR, the second statement identifies the task as object number 211.

   ```
   OPEN (file_variable, 'SY$NET:"TASK=GOOBAR"',HISTORY:=NEW);

   OPEN (file_variable, 'SY$NET:"211="',HISTORY:=NEW);
   ```

2. The following statements instruct the NSP to establish a connection to a task. The first statement identifies the named task as POOBAH; the second statement identifies the task generically as object number 211.

   ```
   OPEN (file_variable, 'MEXICO::"TASK=POOBAH"',HISTORY:=OLD);

   OPEN (file_variable, 'MEXICO::"211="',HISTORY:=OLD);
   ```

9.2.6 Error Returns from I/O Requests

The MicroPower/Pascal kernel recognizes an error condition caused by the execution of an I/O request. Those errors may cause an exception condition, depending on the error-handling policy of your application (see Chapter 17).

The section Error Returns in the description of each request lists the exceptions that are directly associated with it. Each description includes the exception type and code and the error message text displayed by the PASDBG program.

Other exceptions that are not listed may also occur when using these requests. Generally, such exceptions are of the SOFT_IO or HARD_IO type and are reported by the MicroPower/Pascal ACP, the NSP, and device drivers when accessed by the I/O requests. Some examples are "Device off line" and "Unsafe." Those exceptions are described in the *MicroPower/Pascal I/O Services Manual.*

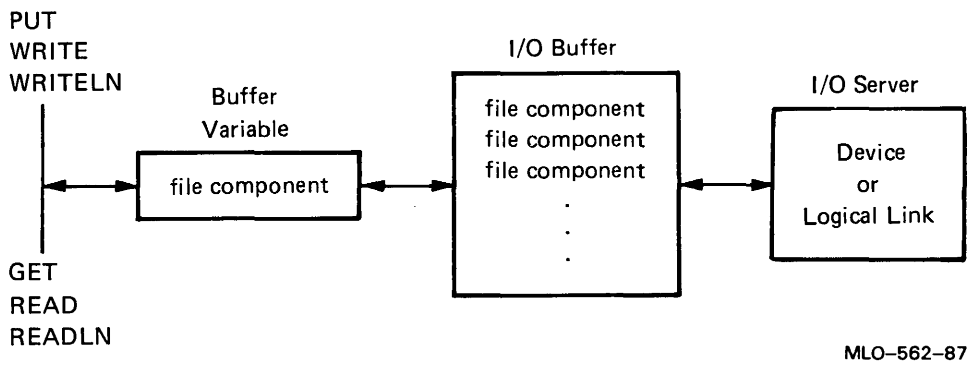
Section 11.2 and Chapter 17 contain additional information about exception handling. See the applicable MicroPower/Pascal messages manual for a more detailed description of each exception.

9.2.7 I/O Server Buffering

Although the physical buffering that takes place between a process and a device is largely transparent, you should be aware of the functioning of the I/O buffer during output operations.

Two levels of buffering exist between the MicroPower/Pascal program and the I/O server: the standard Pascal buffer variable (see Section 2.6) and the I/O buffer. This relationship is shown in Figure 9–1.

Figure 9–1: MicroPower/Pascal Program and I/O Server Buffering Relationship



The I/O buffer exists between the buffer variable and the I/O server and may contain several file components. The I/O buffer is used to group multiple file components together in a single I/O message. A large buffer reduces the number of messages that must be exchanged between the object-time system (OTS) and the I/O server handling the open file.

During input operations, a GET, READ, or READLN obtains a file component from the I/O buffer and places it into the buffer variable. The actual reading from the I/O server to the I/O buffer is automatic and asynchronous with program operation.

During output operations, a PUT, WRITE, or WRITELN places the contents of the buffer variable into the I/O buffer. The writing of the I/O buffer to the I/O server is automatic and asynchronous, with program operation occurring when the buffer is about to overflow. Thus, data output to the I/O server may not always occur when the program requires it. The BREAK and EMPTY_BUFFER requests make sure that the I/O buffer (but not the buffer variable) is emptied.

The OPEN statement lets you specify the I/O buffer size and whether single or double I/O buffering is in effect.

9.2.8 Open and Closed Files

An open file is one that has been identified in a call to the OPEN procedure. Except for the standard files INPUT and OUTPUT, you must initialize the I/O system with the OPEN procedure before accessing a file. Thereafter, you use the RESET procedure to prepare the file for input and the REWRITE procedure to prepare the file for output. When file operations are complete, you close the file with the CLOSE procedure or purge the file with the PURGE procedure. When an open file is closed, buffer operations are finished, and dynamically allocated storage is returned to the heap. If a program (static process) terminates, all files that are open are automatically purged.

9.2.9 Standard Pascal File Variables INPUT and OUTPUT

The standard Pascal file variables INPUT and OUTPUT are predefined as files of type TEXT. They are automatically opened for you and have default characteristics, as if you had specified the following statements in your program:

```
VAR
  FSIZE : INTEGER;
   .
   .
   .
FSIZE := 0;
OPEN ( OUTPUT, 'TTA0:',
  FILESIZE := FSIZE,
  BUFFERSIZE := 66,
  HISTORY := NEW,
  ACCESS_METHOD := SEQUENTIAL,
  OVERLAPPED := ENABLE,
  AUTOEMPTY := TRUE );
REWRITE (OUTPUT);
OPEN ( INPUT, 'TTA0:',
  FILESIZE := FSIZE,
  BUFFERSIZE := 132,
  HISTORY := OLD,
  ACCESS_METHOD := SEQUENTIAL,
  OVERLAPPED := DISABLE,
  AUTOEMPTY := FALSE );
RESET (INPUT);
```

You need not explicitly open INPUT and OUTPUT unless you want to change their operational parameters or assign them to different I/O servers.

Additional default characteristics, such as echoing, are established by the terminal driver's prefix file, as described in the *MicroPower/Pascal I/O Services Manual.*

9.2.10 Additional Files Required for Using the I/O System

Many of the I/O requests described in this chapter depend on symbol definitions and library routines that are external to the MicroPower/Pascal compiler (see Appendix I for details).

9.3 BREAK

The BREAK procedure writes the contents of the I/O buffer to the I/O server. (Ordinarily, the MicroPower/Pascal I/O system performs output to an I/O server only when an output request will cause I/O buffer overflow.) BREAK lets a process synchronize its operation of the completion of output to the I/O server. After a call to BREAK, control does not return to the caller until the I/O system writes the contents of the I/O buffer to the I/O server.

BREAK is a null operation for an input file.

The EMPTY_BUFFER procedure performs an asynchronous BREAK procedure operation for files opened with double buffering (OVERLAPPED:= ENABLE).

Syntax

BREAK (file-variable)

file-variable

The identifier of the file variable associated with the I/O buffer to be written.

When the I/O server is a logical link, BREAK makes the nontext data file being written, using either PUT or WRITE, available to the logical link partner and defines the end of the logical record expected by the partner. This operation is performed automatically for text data files being written to a logical link with WRITELN. See the *MicroPower/Pascal I/O Services Manual* for more information.

Example

```
BREAK(Fvar);
```

Error Returns

See Sections 9.2.6 and 11.2 for general information about error returns. The following exception codes may be returned:

ES$FNO	(type: SOFT_IO)—File not open
ES$FVC	(type: SOFT_IO)—File-variable contention error; two or more processes attempted concurrent access to the same file variable

9.4 CLOSE

The CLOSE procedure disconnects a file variable from an I/O server and returns all buffer space to the heap.

CLOSE provides for an orderly termination of file use. CLOSE permits a process to synchronize its operation on the completion of I/O operations to an I/O server. CLOSE operates synchronously, returning control to the caller only when the operation is complete. (Contrast with the PURGE procedure.)

For output files, CLOSE writes the contents of the I/O buffer to the I/O server. A CLOSE to an output file on a directory-structured I/O server makes the external file permanent if the file was opened with DISPOSITION:=SAVE and HISTORY:=NEW. The file is deleted if DISPOSITION:=DELETE.

A CLOSE to a file opened on a logical link disconnects the caller from the logical link partner and causes EOF to be TRUE for that partner.

Syntax

CLOSE (file-variable)

file-variable
The identifier of the file variable associated with the I/O buffer being closed.

Example

```
CLOSE(Fvar);
```

Error Returns

See Sections 9.2.6 and 11.2 for general information about error returns. The following exception code may be returned:

ES$FNO (type: SOFT_IO)—File not open

9.5 DELETE_FILE

The DELETE_FILE procedure deletes a specified named external file on a directory-structured device.

Syntax

DELETE_FILE (file-specification [[STATUS := status-record]])

file-specification

A character string constant or the identifier of a string variable that specifies the file to be deleted; a standard MicroPower/Pascal external file specification (see Section 9.2.5.1).

status-record

The identifier of a variable of predefined record type EXC_STATUS that may receive an exception type and code. If you specify this parameter, the exception status (either success or error) that results from issuing the request is reported in this variable. Otherwise, an error causes the corresponding exception to be reported. The format of the exception record is described in Section 11.1.2.

Rules and Defaults

- You must make sure that another process is not accessing the specified file.

Examples

1. This example deletes the file identified by a string constant.

   ```
   DELETE_FILE('DYA1:test.dat');
   ```

2. This example deletes a file identified by a string variable. The actual file specification is determined at run time.

   ```
   DELETE_FILE(string_var);
   ```

Error Returns

See Sections 9.2.6 and 11.2 for general information about error returns. The following exception code may be returned:

ES$NMF (type: RESOURCE)—Insufficient space for file variable

9.6 EMPTY_BUFFER

The EMPTY_BUFFER procedure writes the contents of the I/O buffer to the I/O server. (Ordinarily, the MicroPower/Pascal I/O system performs output to an I/O server only when an output request causes a buffer overflow condition.) EMPTY_BUFFER operates asynchronously when a file is opened with double buffering (OVERLAPPED:= ENABLE specified in OPEN).

After a call to EMPTY_BUFFER, control will be returned immediately to the calling process unless the I/O system has not finished writing the contents of the previous buffer to the I/O server. If a file is not opened with double buffering (OVERLAPPED:= DISABLE), the calling process waits until output to the I/O server is complete (identical to the operation of BREAK).

When the I/O server is a logical link, EMPTY_BUFFER makes the nontext data file being written (PUT or WRITE) available to the logical link partner and defines the end of the logical record expected by the partner. This operation is performed automatically for text file data being written to a logical link with WRITELN. See the *MicroPower/Pascal I/O Services Manual* for more information.

EMPTY_BUFFER is a null operation for all input files.

The EMPTY_BUFFER procedure performs an asynchronous break operation for files opened with double buffering (OVERLAPPED:= ENABLE). The BREAK procedure performs a synchronous EMPTY_BUFFER operation.

Syntax

EMPTY_BUFFER (file-variable)

file-variable

The identifier of the file variable associated with the I/O buffer to be written.

Example

```
EMPTY_BUFFER(Fvar);
```

Error Returns

See Sections 9.2.6 and 11.2 for general information about error returns. The following exception codes may be returned:

ES$FNO	(type: SOFT_IO)—File not open
ES$FVC	(type: SOFT_IO)—File-variable contention error; two or more processes attempted concurrent access to the same file variable

9.7 EOF

The EOF function tests the end-of-file (EOF) status and returns a Boolean value. A TRUE value indicates that the buffer variable's position is beyond the last file component. A FALSE value indicates that the buffer variable's position is not beyond the last file component.

Syntax

EOF ⟦ (file-variable) ⟧

file-variable
The identifier of the file variable to be tested. If no file variable is given, INPUT is tested.

Rules and Defaults

- When EOF is TRUE, the content of the buffer variable is undefined.
- When EOF is TRUE, EOLN is also TRUE.
- For files of type TEXT, EOF returns a TRUE when the ASCII code for CTRL/Z is encountered.
- For nontext files, EOF becomes TRUE when the physical end of the file is reached. Since direct-access block replaceable devices are read and written in 512-byte block increments, EOF becomes TRUE when the last block of the file is accessed, not when the last file component is accessed.

  Note

  Because the last component in a file will not reside exactly at the end of the last block, unused storage locations beyond the last file component, though accessible, will not contain meaningful data. Therefore, the calling routine must determine the logical end of file. Two possible ways to do so are by reading the exact number of components that were written to the file or by searching the file for an end-of-file data item.

- Over a logical link, EOF returns FALSE when the logical link is functional. EOF becomes TRUE if the logical link partner issues a CLOSE request for the link's file variable.
- An EOF operation is affected by delayed device access (see Section 9.30).

Examples

1. This program segment displays characters from a file of type TEXT at the console terminal. The input file is assumed to be open and reset.

```
VAR
  Fvar: TEXT;
  Ch: CHAR
BEGIN
  WHILE NOT EOF(fvar) DO
  BEGIN
    Ch:=Fvar^;
    GET(Fvar);
    WRITELN(Ch);
  END;
END;
```

2. This example program loops executing the command file SHOWSYS.COM in the default DECnet account on the node at address 13.5. The program uses PUT rather than WRITE to defeat the AUTOEMPTY feature. AUTOEMPTY is done only after the WRITELN.

```
[SYSTEM(MICROPOWER), DATA_SPACE(2000)] PROGRAM Network_Read;
VAR
  F : TEXT;
  Ch : CHAR;
  I : INTEGER;
  U : UNSIGNED;
BEGIN
  WHILE TRUE DO
  BEGIN
    OPEN(F, '13.5::"TASK=SHOWSYS"', BUFFERSIZE := 100,
         HISTORY := OLD, OVERLAPPED := ENABLE);
    RESET(F);
    WHILE NOT EOF(F) DO
    BEGIN
      WHILE NOT EOLN(F) DO
      BEGIN
        OUTPUT^ := F^;
        PUT(OUTPUT);
        GET(F);
      END;
      WRITELN (OUTPUT);
      GET(F);
    END;
  CLOSE(F);
  END;
END.
```

Error Returns

See Sections 9.2.6 and 11.2 for general information about error returns. The following exception codes may be returned:

ES$FNO	(type: SOFT_IO)—File not open
ES$FVC	(type: SOFT_IO)—File-variable contention error; two or more processes attempted concurrent access to the same file variable

9.8 EOLN

The EOLN function tests for the end-of-line (EOLN) marker (RETURN ASCII character) within a file of type TEXT and returns a Boolean value indicating the result of that test. EOLN returns TRUE when the EOLN marker has been reached; otherwise, EOLN returns FALSE.

Syntax

EOLN ⟦ (file-variable) ⟧

file-variable

The identifier of the file variable to be tested. If no file variable is given, INPUT is tested.

Rules and Defaults

- When EOLN is TRUE, the buffer variable contains a space character.
- An EOLN operation is affected by delayed device access (see Section 9.30).

Example

This program segment reads characters from a file of type TEXT and writes them to the file OUTPUT.

```
VAR  Testfile : TEXT;
     Ch : CHAR;
BEGIN

  WHILE NOT EOF(Testfile) DO
  BEGIN
    WHILE NOT EOLN(Testfile) DO
    BEGIN
      READ(Testfile,Ch);
      WRITE(Ch);
    END;
    READLN(Testfile);
    WRITELN;
  END;
END.
```

Error Returns

See Sections 9.2.6 and 11.2 for general information about error returns. The following exception codes may be returned:

ES$FNO	(type: SOFT_IO)—File not open
ES$FVC	(type: SOFT_IO)—File-variable contention error; two or more processes attempted concurrent access to the same file variable

9.9 FIND

The FIND procedure positions the file on the specified component and assigns the value of that component to the buffer variable. FIND operates only on input files residing on directory- and nondirectory-structured I/O servers when the DIRECT or UPDATE options to OPEN are selected.

Syntax

FIND (file-variable , component-number);

file-variable
The identifier of the file variable of the file to be searched.

component-number
A value specified by a constant, expression, or variable of type INTEGER that is the ordinal displacement of the file component from the beginning of the file. The first component is 1.

Rules and Defaults

- A FIND operation is affected by delayed device access (see Section 9.30).

Examples

1. This example shows the component-number specified by a constant.

```
FIND(Fvar,10);
```

2. This example shows the component-number specified by an expression.

```
FIND(Fvar, index+2);
```

3. This example shows the component-number specified by a variable.

```
FIND(Fvar, offset);
```

Error Returns

See Sections 9.2.6 and 11.2 for general information about error returns. The following exception codes may be returned:

ES$DAS	(type: SOFT_IO)—Direct access on a sequential file
ES$FNO	(type: SOFT_IO)—File not open
ES$FVC	(type: SOFT_IO)—File-variable contention error; two or more processes attempted concurrent access to the same file variable
ES$REF	(type: SOFT_IO)—Read past EOF

9.10 FORMAT_RX02

The FORMAT_RX02 procedure formats a diskette mounted in the specified unit of an RX02 disk drive.

Syntax

FORMAT_RX02 ⟦ (⟦controller⟧, ⟦unit⟧, ⟦density⟧) ⟧;

controller

A string constant or a string variable that identifies the controller of the disk being formatted. The default value is 'A'.

unit

An integer constant or a variable of type INTEGER that specifies the unit of the disk to be formatted. The maximum value is 1; the default is 0.

density

The recording density formatting; either DOUBLE, for double density, or SINGLE, for single density; the default is DOUBLE.

Examples

1. This statement when executed formats a diskette in unit 1 of controller A, using double density.

   ```
   FORMAT_RX02('A',1);
   ```

2. This statement when executed formats a diskette in unit 0 of controller B, using single density.

   ```
   FORMAT_RX02('B',,SINGLE);
   ```

3. This statement when executed formats a diskette in unit 0 of controller A, using double density.

   ```
   FORMAT_RX02;
   ```

Error Returns

See Sections 9.2.6 and 11.2 for general information about error returns. The following exception code may be returned:

ES$DVF	(type: SOFT_IO)—Attempt to signal driver failed; illegal controller parameter or handler not loaded

9.11 GET

The GET procedure positions the file variable on the next component, then assigns the value of that component to the buffer variable.

Syntax

GET (file-variable);

file-variable

The identifier of the file variable associated with the file from which the GET procedure reads the data.

Rules and Defaults

- For a file opened for sequential, direct, or update access, a RESET request must be executed before issuing the first GET.
- A GET operation is affected by delayed device access (see Section 9.30).

Examples

1. This program segment demonstrates the basic use of GET. Given a file identified by the file variable Data, GET's buffer variable would be called Data^. The current value of Data^ is assigned to variable Ch, which is the same type as the components in the file. In a file of TEXT, the components are of type CHAR. The OPEN procedure identifies the file; the RESET procedure prepares the file for input and places the first component in the buffer variable Data^. The WHILE statement establishes a loop condition that the terminal executes after the last component in the file is read. The call to GET reads a new value into Data^. The WRITE procedure sends the character to the file OUTPUT (default file for WRITE).

```
VAR
  Data   : TEXT;
  Ch   : CHAR;
  namstr: NAME_STR;
  namdesc:STRUCTURE_DESC;
BEGIN
  Namstr := 'DLA1  ';
  CREATE_LOGICAL_NAME (LENGTH:=2, STRING:='DK',
                       DESC:=namdesc, NAME:=namstr);
  OPEN(Data, 'DK:IN.TXT', HISTORY:= OLD);
  RESET(Data);
    .
    .
    .
  WHILE NOT EOF(Data) DO
  BEGIN
    Ch := Data^;
    GET(Data);
    WRITE(Ch);
  END;
END.
```

2. This program segment reads real numbers from the file 'DYA0:test.dat' and displays them at the console terminal. The first component in the file is assumed to be the file size, that is, the number of components that the file contains.

```
VAR
  Fvar  : FILE OF REAL;
  Datum  : REAL;
  Count  : INTEGER;
BEGIN
  OPEN(Fvar, 'DYA0:test.dat', HISTORY:= OLD);
  RESET(Fvar);
   .
   .
   .
  BEGIN
    Datum := Fvar^ ;      (* Get the first component which, *)
    Count=ROUND(Datum);   (* in this example, is the number *)
    GET(Fvar);            (* of components in file *)
  END;
  WHILE Count > 0 DO
    BEGIN
      Datum:=Fvar^;       (* Get first file component *)
      GET(Fvar);          (* Point to next component *)
      WRITELN(Datum);     (* Send it to OUTPUT *)
      Count:= Count - 1;  (* Decrement component count *)
    END;
END.
```

Error Returns

See Sections 9.2.6 and 11.2 for general information about error returns. The following exception codes may be returned:

ES$FNR (type: SOFT_IO)—File not reset

ES$REF (type: SOFT_IO)—Read past EOF

ES$FVC (type: SOFT_IO)—File-variable contention error; two or more processes attempted concurrent access to the same file variable

9.12 INIT_DIRECTORY

The INIT_DIRECTORY procedure initializes a directory on the specified directory-structured I/O device.

Syntax

INIT_DIRECTORY (device-specification , ⟦directory-size⟧ , ⟦STATUS := status-record⟧);

device-specification

A string constant or a string variable that specifies the device and unit number containing the media to be initialized; a standard MicroPower/Pascal device specification (see Section 9.2.5.1).

directory-size

An integer constant or variable of type INTEGER in the range 0 to 31 that specifies the size, in segments, of the device's directory. Each segment occupies two blocks. If you specify a value of 0 or do not specify a valid value, the directory is created with the default directory size for that device.

status-record

The identifier of a variable of predefined record type EXC_STATUS that may receive an exception type and code. If you specify this parameter, the exception status (either success or error) that results from issuing the request is reported in this variable. Otherwise, an error causes the corresponding exception to be reported. The format of the exception record is described in Section 11.1.2.

Rules and Defaults

- The default size depends on the media storage capacity shown below:

Media Size	Number of Segments
< 512	1
> 512	4
> 2048	16
> 12,288	31

  The *MicroPower/Pascal I/O Services Manual* describes the structure of device directories, number of files in a segment, and the size of specific device media.

- You must make sure that another process is not accessing the specified device.

Examples

1. This statement when executed initializes the directory on DYA0:, using the default directory size for that device.

   ```
   INIT_DIRECTORY('DYA0:');
   ```

2. This statement when executed initializes the directory on DYA1:, using the value of the variable size as the number of segments in the device's directory.

   ```
   INIT_DIRECTORY('DYA1:',size);
   ```

Error Returns

See Sections 9.2.6 and 11.2 for general information about error returns. The following exception codes may be returned:

None

9.13 OPEN

The OPEN procedure prepares the I/O system to access a specified file by associating a file variable with an I/O server and establishing the file's characteristics and access parameters.

A file that has been identified in a call to OPEN is said to be opened. You must explicitly use the OPEN procedure on all files except when using the default characteristics of INPUT and OUTPUT (see Section 9.2.9) and when using the default temporary file created by the REWRITE procedure. Upon completion of a call to OPEN, EOF is undefined. Once a file is opened, you use the RESET procedure to prepare an open file for input or update access and the REWRITE procedure to prepare the file for output access.

Syntax

```
OPEN ( file-variable
       i/o-specification
       [[ STATUS := status-record ]]
       [[ FILESIZE := file-size ]]
       [[ BUFFERSIZE := buffer-size ]]
       [[ HISTORY := { NEW
                       OLD } ]]
       [[ ACCESS_METHOD := { DIRECT
                             SEQUENTIAL
                             UPDATE } ]]
       [[ DISPOSITION := { SAVE
                           DELETE } ]]
       [[ OVERLAPPED := { ENABLE
                          DISABLE } ]]
       [[ AUTOEMPTY := { TRUE
                         FALSE } ]] )
```

file-variable

The identifier of the file variable to assign to the opened file. This file variable is the logical (internal) name of the file and is used by a program to refer to the open file. If the file variable is not specified, a compilation error occurs.

i/o-specification

A string constant or the identifier of a string variable or a conformant array that specifies a standard MicroPower/Pascal I/O specification (see Section 9.2.5.1).

status-record

The identifier of a variable of predefined record type EXC_STATUS that may receive an exception type and code. If you specify this parameter, the exception status (either success or error) that results from issuing the request is reported in this variable. Otherwise, an error causes the corresponding exception to be reported. The format of the exception record is described in Section 11.1.2.

file-size

The identifier of a variable of type INTEGER that contains a value that is the amount of storage occupied by a file residing on a directory-structured I/O device. For a new file (HISTORY:= NEW), this parameter specifies the amount of storage to allocate for the file. The value of the file size may be one of the following:

- An integer value that is the number of 512-byte blocks to allocate
- 0, the default, which requests the I/O system to allocate half of the largest free space or all of the second largest space, whichever is larger
- –1, which requests the I/O system to allocate the largest free space

On return to the caller, for both new and old files, the variable will contain an integer value that is the amount of storage space actually allocated.

buffer-size

An integer constant that is the suggested size, in bytes, of the I/O buffer to use for this file. The default value is 1. The maximum buffer-size is 7680 bytes for direct-access block-replaceable devices and 8128 for other devices. This restriction results from the architecture of the PDP–11 systems, particularly the maximum length of a page. The 8128 limit results from subtracting the maximum relocatable offset of 64 bytes from the maximum length of a page (8192 bytes). The 7680 limit is the largest 512 multiple under the 8128 limitation, that is, (15 * 512).

If the I/O server is a direct-access block-replaceable device, the I/O system adjusts the buffer size you specify as follows:

- If the value you specify is not a multiple of 512, a buffer size is selected that is the smallest multiple of 512 larger than the specified value.
- If you do not specify a value, the default is 512.

If the I/O server is not a direct-access device, the size you select is used unless the I/O server provides the I/O system with a value to override the size you specify. See the *MicroPower/Pascal I/O Services Manual* for buffer-size information for specific I/O servers.

HISTORY

Establishes the I/O status of the file. NEW declares an output file. OLD, the default, declares an input file.

- NEW—For directory-structured I/O servers, enters the name of a new file in the device's directory. A new file with the same name as an existing file replaces the existing file in the directory when the file is closed, unless the file was protected (see PROTECT_FILE request).
- For logical-link I/O servers, declares that the caller is the passive task identified by the i/o-specification parameter.
- OLD (default)—For directory-structured I/O servers, searches the device's directory for the specified file. Since old (existing) external files retain the same space allocation as when created, they cannot increase in size.

For logical-link I/O servers, declares to the NSP that the caller is an active task and requests that the NSP search for a passive task identified by the i/o-specification parameter.

ACCESS_METHOD

The order in which components of a file can be accessed. The options are DIRECT, SEQUENTIAL, or UPDATE; the default is SEQUENTIAL.

- DIRECT—Components of the file can be read randomly with the FIND procedure. This option is valid only for old files (HISTORY:=OLD) that reside on directory- or nondirectory-structured block-replaceable I/O servers. DIRECT implicitly sets the OVERLAPPED parameter to DISABLE. You cannot use the DIRECT method to access a text file.
- SEQUENTIAL (default)—Components of the file will be accessed sequentially.
- UPDATE—Components of the file can be read, updated, and written back into the file. The FIND procedure can be used to access the components of such a file at random. This option is valid only for old files (HISTORY:=OLD) that reside on directory- or nondirectory-structured block-replaceable I/O devices and implicitly sets the OVERLAPPED parameter to DISABLE. You cannot use the UPDATE method to access a text file.

DISPOSITION

Controls whether an output file on a directory-structured I/O device is to be temporary or permanent. This parameter is meaningful only for new files (HISTORY := NEW). The values DELETE or SAVE can be assigned; the default is SAVE.

- SAVE (default)—The output file is kept as a permanent file when the file is closed.
- DELETE—The output file is temporary. The directory entry is deleted when the file is closed.

OVERLAPPED

Controls whether double buffering is in effect for the file. The options are ENABLE or DISABLE; the default is DISABLE. (The size of the buffers is specified by the buffer-size parameter.)

- ENABLE—Specifies that I/O transfers will be double buffered. This parameter is ignored for an I/O server that is a ring buffer or when the access method is DIRECT or UPDATE.
- DISABLE (default)—Specifies that I/O transfers will not be double buffered.

AUTOEMPTY

Controls whether the EMPTY_BUFFER request is automatically issued after each WRITE or WRITELN request. The options are TRUE or FALSE; the default is FALSE.

- TRUE—Specifies that the EMPTY_BUFFER procedure be called automatically after each WRITE or WRITELN request.
- FALSE (default)—Specifies that the contents of a buffer will be written when a WRITE or WRITELN request would cause buffer overflow. Section 9.2.7 describes I/O buffering.

Restriction

The file system makes only rudimentary checks before believing the data in a disk's directory segments. Certain uninitialized flexible diskettes pass those few checks and are not flagged as uninitialized.

Thus, an OPEN (with a STATUS parameter) of an uninitialized flexible diskette may not return an error status indication. Ultimately, the file system will incur an exception, such as a memory management trap (ES$MMU).

Examples

1. This statement when executed opens the file specified by SPEC and assigns it the internal name FILVAR. The file size is the value of the variable fsize. The remaining parameters use the default values.

   ```
   OPEN( FILVAR, SPEC, FILESIZE:=fsize);
   ```

2. This statement when executed opens the file specified by the string constant COMMX2::"TASK=FRED" and assigns it the internal name FILVAR. I/O transfers are double buffered and the file size is the value of the variable fsize. The remaining parameters use the default values.

   ```
   OPEN( FILVAR, 'COMMX2::"TASK=FRED"', HISTORY:=OLD,
   OVERLAPPED:=ENABLE, FILESIZE:=fsize);
   ```

3. This statement when executed opens the input buffer on TTA0: as a new file and assigns it the logical name FILVAR. The remaining parameters use the default values.

   ```
   OPEN( FILVAR, 'TTA0:', HISTORY:=NEW);
   ```

4. This statement when executed opens the ring buffer RINGBF: as an old file and assigns it the logical name INPUT. The EMPTY_BUFFER request is automatically issued after each WRITE or WRITELN request. The remaining parameters use the default values.

   ```
   OPEN(INPUT, 'RINGBF:', AUTOEMPTY:=TRUE);
   ```

5. This statement when executed opens the file specified by SPEC and assigns it the logical name FILVAR. The file size is the value of fsize and the I/O buffer size is the value of buffsize. The file components can be read randomly with the FIND procedure. The remaining parameters use the default values.

   ```
   OPEN(FILVAR, SPEC, FILESIZE:=fsize, BUFFERSIZE:=buffsize,
   HISTORY:=OLD, ACCESS_METHOD:=DIRECT);
   ```

Error Returns

See Sections 9.2.6 and 11.2 for general information about error returns. The following exception codes may be returned:

ES$FAO	(type: SOFT_IO)—File already open
ES$IUP	(type: SOFT_IO)—Illegal use of UPDATE parameter
ES$NMB	(type: RESOURCE)—Insufficient space for data buffer

The request may return the following error, though not as a result of standard Pascal programming practice.

ES$RSZ	(type: SOFT_IO)—Record size of 0 specified

9.14 PAGE

The PAGE procedure sends a form-feed character to the output file. The file must be of predefined type TEXT.

Syntax

PAGE (file-variable);

file-variable

The identifier of the file variable associated with the output file.

Error Returns

See Sections 9.2.6 and 11.2 for general information about error returns. The following exception codes may be returned:

ES$FNO	(type: SOFT_IO)—File not open
ES$FVC	(type: SOFT_IO)—File-variable contention error; two or more processes attempted concurrent access to the same file variable

9.15 PROTECT_FILE

The PROTECT_FILE procedure protects a named external file residing on a directory-structured I/O device from deletion. This protection can be removed with the UNPROTECT_FILE procedure.

Syntax

PROTECT_FILE (file-specification , ⟦STATUS := status-record⟧);

file-specification

A string constant or the identifier of a string variable that specifies the name of the file to be protected and the device on which the file resides; a standard MicroPower/Pascal file specification (see Section 9.2.5.1).

status-record

The identifier of a variable of predefined record type EXC_STATUS that may receive an exception type and code. If you specify this parameter, the exception status (either success or error) that results from issuing the request is reported in this variable. Otherwise, an error causes the corresponding exception to be reported. The format of the exception record is described in Section 11.1.2.

Rules and Defaults

- Read and write operations can be performed on a protected file.
- You must ensure that another process is not accessing the specified device.

Example

```
PROTECT_FILE('DYA0:Exampl.dat');
```

Error Returns

See Sections 9.2.6 and 11.2 for general information about error returns. The following exception codes may be returned:

None

9.16 PURGE

The PURGE procedure disconnects the specified file variable from the I/O device and terminates access to the file. All buffer space allocated to the file is returned to the heap, and the contents of the buffer variable and the I/O buffer are lost. The PURGE operation is asynchronous, and control returns immediately to the caller. (Contrast with the CLOSE procedure.)

A PURGE to an output file opened on a directory-structured I/O server with HISTORY := NEW causes the space that the file occupies to become available for reuse (the file is not made permanent).

A PURGE to a file opened on a logical link causes abnormal termination of the link and may cause an exception for the logical-link partner.

Syntax

PURGE (file-variable);

file-variable

The identifier of the file variable associated with the file to be purged.

Example

```
PURGE(Fvar);
```

Error Returns

See Sections 9.2.6 and 11.2 for general information about error returns. The following exception code may be returned:

ES$FNO (type: SOFT_IO)—File not open

9.17 PUT

The PUT procedure writes the value of the buffer variable to the file and positions the file on the next component.

Syntax

PUT (file-variable);

file-variable

The identifier of the file variable associated with the file to which the PUT procedure writes data.

Rules and Defaults

- A REWRITE request must be executed before issuing the first PUT to a file.
- After execution of a PUT, the value of the buffer variable becomes undefined, unless the file was opened in UPDATE mode.
- Output to an I/O server occurs asynchronously with this request. (Section 9.2.7 describes I/O buffering.)
- A PUT operation is affected by delayed device access (see Section 9.30).

Example

This program segment writes an integer array to a file.

```
VAR
  Fvar         : FILE OF INTEGER;
  Datum        : ARRAY[1..100] OF INTEGER;
  Index        : INTEGER;
  Fsize        : INTEGER;
BEGIN
  Fsize:= 10;
  OPEN(Fvar, 'DYA0:test.dat',FILESIZE:=Fsize, HISTORY:= NEW);
    .
    .
    .
  FOR Index := 1 TO 100 DO
  BEGIN
    Fvar^ := Datum[Index];
    PUT(Fvar);
  END;
```

Error Returns

See Sections 9.2.6 and 11.2 for general information about error returns. The following exception codes may be returned:

ES$FNO	(type: SOFT_IO)—File not open
ES$FNW	(type: SOFT_IO)—File not rewritten
ES$FVC	(type: SOFT_IO)—File-variable contention error; two or more processes attempted concurrent access to the same file variable
ES$WEF	(type: SOFT_IO)—Write past EOF

9.18 READ

The READ procedure reads one or more file components into a variable.

Syntax

READ (⟦file-variable⟧ , {variable-id} ...);

file-variable

The identifier of the file variable associated with the input file. If no file variable is specified, the default is INPUT.

variable-id

The name of the variable into which a file component will be read.

Rules and Defaults

- For nontext files, READ reads the components of any scalar or structured type from the file by performing the following sequence for each variable in the parameter list:

  ```
  variable-identifier := file-variable^;
  GET (file-variable);
  ```

  READ assigns file components to the variables in parameter list order until READ has found a value for each variable. The file components must be assignment compatible with the specified variable.
- While reading text file components into a scalar variable, READ skips any spaces that precede a valid value, then reads the value until a nonnumeric character is encountered.
- For text files, file components to be read into a character or a string variable must not be delimited by spaces because the components are read from the file and assigned to the variable one character at a time.
- For text files, values being read into scalar variables must be assignment compatible with those variables.
- For text files, READ reads the value of any scalar type or string type from the file (includes scalar and string components of record and array types); enumerated and structured types are not allowed.
- For text files, READ performs the assignment and GET sequence shown in item 1 on each file component until READ has read a series of characters that represent a legal value for the type of the next variable in the parameter list. If the variable is of a scalar type, READ converts the value appropriately. The procedure continues to read components until it has assigned a value to each variable in the list.
- Except for string variables, a READ will skip over the end-of-line marker and position the file at the beginning of the next line.
- A READ into a variable of type CHAR when EOLN is TRUE obtains a space character, and the file position advances.
- A READ into a string variable when EOLN is TRUE obtains space characters; a READLN must be issued to advance the position past the end-of-line marker.

- When reading into a string variable, READ assigns successive characters to successive elements of the packed array until it is full. If READ encounters the end of the line before the array is full, the remaining elements are filled with spaces. If the array is filled before the end of the line is reached, the next READ begins with the next character on the same line.

Note

Every nonempty text file ends with an EOLN marker and an EOF marker. Therefore, the function EOF never returns TRUE when you are reading strings with READ. To test for EOF when reading strings, use READLN to advance beyond the EOLN marker.

Examples

1. This program segment reads a value for a variable of type Cube from a file of component type Cube.

```
TYPE
  Cube : RECORD
           X,Y,Z : REAL;
           Weight: REAL;
           Number: INTEGER;
         END;

VAR
  Cubefile : FILE OF Cube;
  Cubic    : Cube;

BEGIN
  OPEN(Cubefile,'DYAO:cubes.dat',HISTORY := OLD);
  RESET(Cubefile);
  READ(Cubefile,Cubic);
```

2. This program segment declares and reads the file Comfile.

```
TYPE
  String = PACKED ARRAY [1..20] OF CHAR;

VAR
  Comfile : TEXT;
  Command : String;
BEGIN
  READ(Comfile,Command);
  READ(Comfile,Command);
```

Comfile contains the following characters:

Run Process 3 Halt Process 1 <EOLN>

The first execution of the READ procedure assigns to the variable Command the value 'Run Process 3 '. The second call to READ assigns the value 'Halt Process 1 ' to Command.

Error Returns

See Sections 9.2.6 and 11.2 for general information about error returns. The following exception codes may be returned:

ES$BIV	(type: SOFT_IO)—Illegal Boolean value
ES$FIV	(type: SOFT_IO)—Illegal floating-point value
ES$FNO	(type: SOFT_IO)—File not open
ES$FVC	(type: SOFT_IO)—File-variable contention error; two or more processes attempted concurrent access to the same file variable
ES$IIV	(type: SOFT_IO)—Illegal integer value
ES$ILV	(type: SOFT_IO)—Illegal long-integer value
ES$UIV	(type: SOFT_IO)—Illegal unsigned value

9.19 READLN

The READLN procedure reads lines of data from a text file.

Syntax

$$\text{READLN} \left[\!\!\left[(\left\{ \begin{array}{l} \text{file-variable} \\ \text{variable-id} \end{array} \right\} , [\![\{\text{variable-id}\} \ldots]\!]) \right]\!\!\right]$$

file-variable
The identifier of a variable of type TEXT associated with the file to be read. If no file variable is specified, the default is INPUT.

variable-id
The name of the variable into which a value will be read.

Rules and Defaults

- READLN performs the following sequence for each variable in the parameter list:

  ```
  READ (file-variable, variable-identifier)
  ```

  READLN performs this sequence until READLN has read a series of characters that represent a legal value for the type of the next variable in the parameter list. If the variable is of a scalar type, READLN converts the value appropriately. The procedure continues to read file components until it has assigned a value to each variable in the list. After reading values for all the listed variables, READLN skips any characters remaining on the current line and positions the file at the beginning of the next line.
- With the file positioned at the end of a line, a call to READLN obtains the first value in the next line.
- When no variable-id is supplied, READLN skips to the next line in the input file.
- After execution, READLN sets EOLN to FALSE except when the next line is empty.

Examples

1. This program segments reads lines containing a 20-character string and a Boolean value.

   ```
   TYPE
     String = PACKED ARRAY [1..20] OF CHAR;
   VAR
     Conditions : TEXT;
     Ready : BOOLEAN;
     Name  : String;
   BEGIN
     READLN( Conditions, Name, Ready );
   ```

   The input file CONDITIONS might look like this:

   ```
   MACHINE NUMBER ONE     TRUE
   ```

2. The statement below skips over an input line of the default file INPUT.

   ```
   READLN;
   ```

3. The statement below skips an input line in the file FVAR.

```
READLN( Fvar );
```

Error Returns

See Sections 9.2.6 and 11.2 for general information about error returns. The following exception codes may be returned:

ES$BIV	(type: SOFT_IO)—Illegal Boolean value
ES$FIV	(type: SOFT_IO)—Illegal floating-point value
ES$FNO	(type: SOFT_IO)—File not open
ES$FVC	(type: SOFT_IO)—File-variable contention error; two or more processes attempted concurrent access to the same file variable
ES$IIV	(type: SOFT_IO)—Illegal integer value
ES$ILV	(type: SOFT_IO)—Illegal long-integer value
ES$UIV	(type: SOFT_IO)—Illegal unsigned value

9.20 Input Integer Conversion Functions (VMS only)

MicroPower/Pascal software supplies the predeclared functions BIN, HEX, and OCT to perform radix conversion after input by the READ and READLN procedures. Those functions operate on INTEGER, UNSIGNED, and LONG_INTEGER values and are legal only for files of type TEXT. A compilation error occurs if you specify those functions for a file that is not of type TEXT.

Note

The input integer conversion functions are only available with MicroPower/Pascal–VMS.

9.20.1 BIN(x)

The BIN function converts the binary representation of a number in a TEXT file into the INTEGER, UNSIGNED, or LONG_INTEGER type of a READ or READLN procedure parameter (x).

Examples

1. The BIN function converts the binary representation of a number (1010101010101010) in a TEXT file into the READ procedure integer parameter (513).

```
READ ( BIN (I) ); (* where I is declared as INTEGER. *)
```

2. The BIN function converts the binary representation of a number (1000000000001010) in a TEXT file into the READ procedure unsigned parameter (32778).

```
READ ( BIN (U) ); (* where U is declared as UNSIGNED. *)
```

3. The BIN function converts the binary representation of a number (00000000000000100000000000000000) in a TEXT file into the READ procedure long integer parameter (131072).

   ```
   READ ( BIN (L) ); (* where L is declared as LONG_INTEGER. *)
   ```

Error Returns

See Sections 9.2.6 and 11.2 for general information about error returns. The following exception codes may be returned:

ES$ILV	(type: SOFT_IO)—Illegal long-integer value
ES$UIV	(type: SOFT_IO)—Illegal unsigned value

9.20.2 HEX(x)

The HEX function converts the hexadecimal representation of a number in a TEXT file into the INTEGER, UNSIGNED, or LONG_INTEGER type of a READ or READLN procedure parameter (x).

Examples

1. The HEX function converts the hexadecimal representation of a number (A) into the READ procedure integer parameter (10).

   ```
   READ ( HEX (I) ); (* where I is declared as INTEGER. *)
   ```

2. The HEX function converts the hexadecimal representation of a number (800A) into the READ procedure unsigned parameter (32778).

   ```
   READ ( HEX (U) ); (* where U is declared as UNSIGNED. *)
   ```

3. The HEX function converts the hexadecimal representation of a number (00040000) into the READ procedure long integer parameter (262144).

   ```
   READ ( HEX (L) ); (* where L is declared as LONG_INTEGER. *)
   ```

Error Returns

See Sections 9.2.6 and 11.2 for general information about error returns. The following exception codes may be returned:

ES$ILV	(type: SOFT_IO)—Illegal long-integer value
ES$UIV	(type: SOFT_IO)—Illegal unsigned value

9.20.3 OCT(x)

The OCT function converts the octal representation of a number in a TEXT file into the INTEGER, UNSIGNED, or LONG_INTEGER type of a READ or READLN procedure parameter (x).

Examples

1. The OCT function converts the octal representation of a number (12) into the READ procedure integer parameter (10).

   ```
   READ ( OCT (I) ); (* where I is declared as INTEGER. *)
   ```

2. The OCT function converts the octal representation of a number (100012) into the READ procedure unsigned parameter (32778).

   ```
   READ ( OCT (U) ); (* where U is declared as UNSIGNED. *)
   ```

3. The OCT function converts the octal representation of a number (400000) into the READ procedure long integer parameter (131072).

   ```
   READ ( OCT (L) ); (* where L is declared as LONG_INTEGER. *)
   ```

Error Returns

See Sections 9.2.6 and 11.2 for general information about error returns. The following exception codes may be returned:

ES$ILV	(type: SOFT_IO)—Illegal long-integer value
ES$UIV	(type: SOFT_IO)—Illegal unsigned value

9.21 RENAME_FILE

The RENAME procedure changes the name of a named external file residing on a directory-structured I/O device.

Syntax

RENAME_FILE (old-name , new-name , ⟦STATUS := status-record⟧);

old-name

A string constant or a string variable that specifies the current name of the file; a standard MicroPower/Pascal external file specification (see Section 9.2.5.1).

new-name

A string constant or a string variable that specifies the new name for the file; a standard MicroPower/Pascal file specification (see Section 9.2.5.1).

status-record

The identifier of a variable of predefined record type EXC_STATUS that may receive an exception type and code. If you specify this parameter, the exception status (either success or error) that results from issuing the request is reported in this variable. Otherwise, an error causes the corresponding exception to be reported. The format of the exception record is described in Section 11.1.2.

Rules and Defaults

- If a file having the same name as the new file name already exists on the device, that file is deleted.
- You must make sure that another process is not accessing the specified device.

Example

```
RENAME_FILE('DYA0:test.dat','DYA0:test.old');
```

Error Returns

See Sections 9.2.6 and 11.2 for general information about error returns. The following exception codes may be returned:

None

9.22 RESET

The RESET procedure readies a file for input by setting EOF to FALSE, positioning the file on its first component, and placing that component in the buffer variable.

Syntax

RESET (file-variable);

file-variable

The identifier of the file variable associated with the file being reset for input.

Rules and Defaults

- An existing output file remains open. Any data in the buffer variable and the I/O buffer is written to the file before positioning it at the beginning.
- If the file is opened as NEW on a directory-structured I/O device, the size is truncated to the amount of data in the file. If the file is opened as OLD, the size remains as specified when opened.
- A RESET operation is affected by delayed device access (see Section 9.30).

Example

```
RESET( Fvar );
```

Error Returns

See Sections 9.2.6 and 11.2 for general information about error returns. The following exception codes may be returned:

ES$FNO	(type: SOFT_IO)—File not open
ES$FVC	(type: SOFT_IO)—File-variable contention error; two or more processes attempted concurrent access to the same file variable
ES$WEF	(type: SOFT_IO)—Write past EOF

9.23 REWRITE

The REWRITE procedure readies a file for output and sets EOF to TRUE.

Syntax

REWRITE (file-variable);

file-variable

The identifier of the file variable associated with the output file.

Rules and Defaults

- REWRITE positions a file on a mass-storage device at the beginning.
- If you do not use the OPEN procedure on a file, REWRITE creates a temporary file on the device associated with the default logical name DK: (see CREATE_LOGICAL_NAME in Chapter 20).
- You may use REWRITE on a file variable any number of times.
- An attempt to REWRITE an open file that was previously RESET will cause the ES$FRO exception if the file was opened with double buffering (OVERLAPPED := ENABLE).

Example

```
REWRITE(Fvar);
```

Error Returns

See Sections 9.2.6 and 11.2 for general information about error returns. The following exception codes may be returned:

ES$FRO	(type: SOFT_IO)—No write access allowed
ES$FVC	(type: SOFT_IO)—File-variable contention error; two or more processes attempted concurrent access to the same file variable

9.24 SQUEEZE_DIRECTORY

The SQUEEZE_DIRECTORY procedure consolidates the directory entries and all unused blocks on the device you specify into a single area.

The operation does not affect the device's bootstrap blocks or files with .BAD file types. This feature prevents you from reallocating bad blocks to new files. (You create files with the .BAD type to mark defective areas on the device media.)

Syntax

SQUEEZE_DIRECTORY (device-specification , ⟦STATUS := status-record⟧);

device-specification

A string constant or a string variable that specifies the device and unit number; a standard MicroPower/Pascal device specification (see Section 9.2.5.1).

status-record

The identifier of a variable of predefined record type EXC_STATUS that may receive an exception type and code. If you specify this parameter, the exception status (either success or error) that results from issuing the request is reported in this variable. Otherwise, an error causes the corresponding exception to be reported. The format of the exception record is described in Section 11.1.2.

Example

```
SQUEEZE_DIRECTORY ( 'DK:' );
```

Error Returns

See Sections 9.2.6 and 11.2 for general information about error returns. The following exception codes may be returned:

None

Application Notes

SQUEEZE_DIRECTORY is a potentially destructive operation if you do not observe the following suggestions:

- The squeeze operation should not be performed if there are files open on the device.
- No new files should be opened during a squeeze operation.
- You must make sure that no other process is accessing the specified device.

9.25 UNPROTECT_FILE

The UNPROTECT_FILE procedure removes the protection from a named external file residing on a directory-structured I/O device. The procedure reverses the action performed by the PROTECT_FILE procedure. You can delete an unprotected file with the DELETE_FILE procedure.

Syntax

UNPROTECT_FILE (file-specification , ⟦STATUS := status-record⟧);

file-specification

A string constant or a string variable specifying the file to be unprotected; a standard MicroPower/Pascal file specification (see Section 9.2.5.1).

status-record

The identifier of a variable of predefined record type EXC_STATUS that may receive an exception type and code. If you specify this parameter, the exception status (either success or error) that results from issuing the request is reported in this variable. Otherwise, an error causes the corresponding exception to be reported. The format of the exception record is described in Section 11.1.2.

Rules and Defaults

- You must make sure that another process is not accessing the specified file.

Examples

1. This statement when executed removes the protection from the file test.dat on DYA1:.

   ```
   UNPROTECT_FILE('DYA1:test.dat');
   ```

2. This program segment when executed removes the protection from the file specified by the variable filespec.

   ```
   VAR filespec : PACKED ARRAY[1..24] OF CHAR;
   .
   .
   .
   UNPROTECT_FILE(filespec);
   ```

Error Returns

See Sections 9.2.6 and 11.2 for general information about error returns. The following exception codes may be returned:

None

9.26 WRITE

The WRITE procedure assigns data to a file.

Syntax

WRITE (⟦file-variable⟧ , {expression} ...);

file-variable

The identifier of the file variable associated with the output file. If no file variable is specified, the default is OUTPUT.

expression

The compile-time or run-time expression value to be written. Values are written with a default field width (see Section 9.28).

Rules and Defaults

- Scalar values must be delimited by spaces to let READ and READLN access them properly. Spaces should not be used to delimit strings, as they become part of the string, when read.
- For nontext files, WRITE writes the value of any scalar or structured type by performing the following sequence for each value in the parameter list:

  ```
  file-variable^ := expression;
  PUT (file-variable);
  ```

  The type of each output value must be assignment compatible with the component type of the file.
- For text files, WRITE writes the value of any scalar type or string type to the file (includes scalar and string components of record and array types and conformant arrays). Enumerated and structured types are not allowed. WRITE converts the value of each expression to a sequence of characters, repeating the assignment and PUT process until all the values in the parameter list have been written to the file.

Note

Writing conformant arrays is only available in MicroPower/Pascal–VMS.

- Output to an I/O server will occur asynchronously with this request. (Section 9.2.7 describes buffering.)

Examples

1. The file Test_scores contains test data that includes the test parameter name, test score value, and score weight factor value. The WRITE procedure writes the values Par_name, Score, a space (""), and Weight_factor into the file in order. The space is used as a delimiter between values to allow READ or READLN to read the values into appropriate scalar variables.

```
VAR
   Test_scores : TEXT;
   Par_name : PACKED ARRAY [1..20] OF CHAR;
   Score : 0..100;
   Weight_factor : REAL;
         .
         .
         .
WRITE (Test_scores, Par_name, Score,"",Weight_factor);
```

2. This example shows the use of string conformant arrays (packed arrays of CHAR with any lower and upper bounds). When executed, the code would output:

   ```
   ABCDEFGHIJ
   ABCDEFGHIJKLMNO
   That's all.
   ```

```
CONST
  A_LOWER_LIMIT = 3;
  A_UPPER_LIMIT = 12;
  B_LOWER_LIMIT = 41;
  B_UPPER_LIMIT = 55;

VAR
  A_ARRAY : PACKED ARRAY [A_LOWER_LIMIT..A_UPPER_LIMIT] OF CHAR;
  B_ARRAY : PACKED ARRAY [B_LOWER_LIMIT..B_UPPER_LIMIT] OF CHAR;
  M : INTEGER;

  PROCEDURE P (VAR C : PACKED ARRAY [i..j : INTEGER] OF CHAR);
  BEGIN
    WRITELN(C)
  END;

BEGIN
  FOR M := A_LOWER_LIMIT TO A_UPPER_LIMIT DO
    A_ARRAY[M] := CHR(65 + M - A_LOWER_LIMIT);
  P(A_ARRAY);

  FOR M := B_LOWER_LIMIT TO B_UPPER_LIMIT DO
    B_ARRAY[M] := CHR(65 + M - B_LOWER_LIMIT);
  P(B_ARRAY);

  WRITELN('That''s all.')
END.
```

Error Returns

See Sections 9.2.6 and 11.2 for general information about error returns. The following exception codes may be returned:

ES$FNO	(type: SOFT_IO)—File not open
ES$FVC	(type: SOFT_IO)—File-variable contention error; two or more processes attempted concurrent access to the same file variable
ES$IFW	(type: SOFT_IO)—Illegal field width

9.27 WRITELN

The WRITELN procedure writes a line of data to a text file and starts a new line.

Syntax

$$\text{WRITELN} \left[\!\!\left[\;(\begin{Bmatrix} \text{file-variable} \\ \text{expression} \end{Bmatrix} [\![\{\text{expression}\} \ldots]\!])\; \right]\!\!\right] ;$$

file-variable

The identifier of the file variable associated with the output file. If no file variable is specified, the default is OUTPUT.

expression

The compile-time or run-time expression value to be written. Values are written with a default field width (see Section 9.28).

Rules and Defaults

- WRITELN writes the value of any scalar type or string type to the file (includes scalar and string components of record and array types and conformant arrays); enumerated and structured types are not allowed.
- Scalar values must be delimited by spaces to allow READ and READLN to access them properly. Spaces should not be used to delimit strings, as they become part of the string when read.
- If no data values are specified, WRITELN writes an end-of-line marker and positions the file at the beginning of the next line.
- WRITELN performs the following sequence for each value in the parameter list:

  ```
  WRITE (file-variable, expression);
  ```

  WRITELN converts the value of each expression to a sequence of characters, writes each value in parameter list order into the text file, inserts an end-of-line marker after the end of the last value, and positions the file at the beginning of the next line.
- Output to an I/O server will occur asynchronously with this request. (Section 9.2.7 describes I/O buffering.)
- When writing over a logical link, WRITELN performs an implicit EMPTY_BUFFER operation to send the line to the logical link partner.

Examples

1. This statement when executed writes a 2 in DATA[1].

   ```
   WRITELN (DATA[1]:2);
   ```

2. This statement when executed writes a blank line in OUTPUT.

   ```
   WRITELN;
   ```

3. This statement when executed writes the specified phrase in TEXTFILE.

   ```
   WRITELN (TEXTFILE,'Analysis begun');
   ```

4. See Section 9.26, Example 2 for an example using a conformant array.

Error Returns

See Sections 9.2.6 and 11.2 for general information about error returns. The following exception codes may be returned:

ES$FNO	(type: SOFT_IO)—File not open
ES$FVC	(type: SOFT_IO)—File-variable contention error; two or more processes attempted concurrent access to the same file variable
ES$IFW	(type: SOFT_IO)—Illegal field width

9.28 Text File Output Field Width Specifications

The output values of a WRITE or WRITELN procedure can be compile-time or run-time expressions with values of any ordinal (excluding enumerated), real, or string type. Each value is written with a default field width that specifies the minimum number of characters to be written for the value.

You can override these default field widths for a particular value by using the following format:

expression : minimum : fraction

expression
A WRITE or WRITELN procedure output value.

minimum
A positive or zero-valued integer expression that specifies the minimum number of characters to be written for the value.

fraction
A positive or zero-valued integer expression that specifies the number of digits to be written to the right of the decimal point for a real value. The output data representation is fixed-point decimal.

Rules and Defaults

- The default field width for each type of output value is:

Data Type	Number of Characters
INTEGER	7
LONG_INTEGER	11
REAL	15
CHAR	1
BOOLEAN	5
UNSIGNED	7

- The default field width for a subrange type is the same as that of the parent type of the subrange.
- The default display format for type REAL values is floating-point format (see Section 2.2). Each value is preceded by one space.
- If the field width is larger than the value, the value is right justified in the field with unused character positions filled with spaces.
- When a value of type PACKED ARRAY OF CHAR is larger than its field-width specification, the excess characters to the right are truncated.
- When a value of type REAL is larger than its field-width specification, the excess characters to the right are truncated, and the last digit is rounded up.
- When a value of type INTEGER, LONG_INTEGER, or UNSIGNED is larger than its field-width specification, the field width is expanded to include all characters in the value.

Examples

```
[ SYSTEM (MICROPOWER), PRIORITY (20),
  DATA_SPACE (1000), STACK_SIZE (200) ] PROGRAM FORMAT;

VAR
 R : REAL;
 I : INTEGER;
 B : BOOLEAN;
 S : PACKED ARRAY [1..5] OF CHAR;
 L : LONG_INTEGER;

BEGIN
  R := 1.555;
  I := -12345;
  B := FALSE;
  S := 'ABCDE';
  L := 145645;
```

```
      (*  FIELD WIDTH                  OUTPUT      *)
      (*  SPECIFICATION              1234567890123  *)

  WRITELN ('*', R, '*');        (*  * 1.5550000E+00*  *)
  WRITELN ('*', I, '*');        (*  * -12345*         *)
  WRITELN ('*', B, '*');        (*  *FALSE*           *)
  WRITELN ('*', S, '*');        (*  *ABCDE*           *)
  WRITELN ('*', L, '*');        (*  *     145645*     *)
  WRITELN ('*', R:3, '*');      (*  *  1.6E+00*       *)
  WRITELN ('*', I:3, '*');      (*  *-12345*          *)
  WRITELN ('*', B:3, '*');      (*  *FAL*             *)
  WRITELN ('*', S:3, '*');      (*  *ABC*             *)
  WRITELN ('*', L:3, '*');      (*  *145645*          *)
  WRITELN ('*', R:10, '*');     (*  * 1.5550E+00*     *)
  WRITELN ('*', I:10, '*');     (*  *    -12345*      *)
  WRITELN ('*', B:10, '*');     (*  *     FALSE*      *)
  WRITELN ('*', S:10, '*');     (*  *     ABCDE*      *)
  WRITELN ('*', L:15, '*');     (*  *         145645* *)
  WRITELN ('*', R:3:1, '*');    (*  * 1.6*            *)
  WRITELN ('*', R:4:2, '*');    (*  * 1.55*           *)
  WRITELN ('Done');             (*  Done              *)
END.
```

9.29 Output Integer Conversion Functions

MicroPower/Pascal software supplies the predeclared functions BIN, HEX, and OCT to perform radix conversion in preparation for output by the WRITE and WRITELN procedures. Those functions operate on INTEGER, UNSIGNED, and LONG_INTEGER values and are legal only for files of type TEXT. A compilation error occurs if you specify those functions for a file that is not of type TEXT.

9.29.1 BIN(x)

The BIN function converts a WRITE or WRITELN procedure parameter (x) of type INTEGER or UNSIGNED to its 16-bit binary representation and of type LONG_INTEGER to its 32-bit binary representation.

Examples

1. The BIN function converts the value of the actual parameter %O'125252' to its binary representation and returns this value to the WRITE procedure. The result would be displayed as 1010101010101010.

   ```
   WRITE ( BIN (%O'125252') );
   ```

2. The BIN function converts the decimal value 10 to its binary representation and returns this value to the WRITE procedure. The result would be displayed as 1010.

   ```
   WRITE ( BIN ( 10 ) );
   ```

3. The BIN function converts the hexadecimal value 20000 to its binary representation and returns this value to the WRITE procedure. The result would be displayed as 100000000000000000.

   ```
   WRITE ( BIN ( %X'20000' ) );
   ```

Error Returns

See Sections 9.2.6 and 11.2 for general information about error returns. The following exception codes may be returned:

ES$IFW (type: SOFT_IO)—Illegal field width

ES$LDZ (type: SOFT_IO)—Long integer divided by zero

9.29.2 HEX(x)

The HEX function converts a WRITE or WRITELN procedure parameter (x) of type INTEGER or UNSIGNED to a 4-digit hexadecimal representation and of type LONG_INTEGER to an 8-digit hexadecimal representation.

Examples

1. The HEX function converts the value of the actual parameter %O'125252' to its hexadecimal representation and returns this value to the WRITE procedure. The result would be displayed as AAAA.

   ```
   WRITE ( HEX (%O'125252') );
   ```

2. The HEX function converts the value of the actual parameter 262144 to its hexadecimal representation and returns this value to the WRITE procedure. The result would be displayed as 40000.

   ```
   WRITE ( HEX (262144) );
   ```

Error Returns

See Sections 9.2.6 and 11.2 for general information about error returns. The following exception codes may be returned:

ES$IFW (type: SOFT_IO)—Illegal field width

ES$LDZ (type: SOFT_IO)—Long integer divided by zero

9.29.3 OCT(x)

The OCT function converts a WRITE or WRITELN procedure parameter (x) of type INTEGER or UNSIGNED to a 6-digit octal representation and of type LONG_INTEGER to an 11-digit octal representation.

Examples

1. The OCT function converts the value of the actual parameter %X'AAAA' to its octal representation and returns this value to the WRITE procedure. The result displayed is the octal value 125252.

   ```
   WRITE ( OCT (%X'AAAA') );
   ```

2. The OCT function converts the value of the actual parameter 131072 to its octal representation and returns this value to the WRITE procedure. The result displayed is the octal value 400000.

   ```
   WRITE ( OCT (131072) );
   ```

Error Returns

See Sections 9.2.6 and 11.2 for general information about error returns. The following exception codes may be returned:

ES$IFW	(type: SOFT_IO)—Illegal field width
ES$LDZ	(type: SOFT_IO)—Long integer divided by zero

9.30 Delayed Device Access

The standard Pascal language definition requires that a file's buffer variable contain the next file component that will be processed by the program. This definition can cause problems when the input data to the program depends on the output data most recently generated, as when printing a message that prompts for data at a terminal. To alleviate such problems in the processing of the text files, MicroPower/Pascal uses a technique called delayed device access, or "lazy lookahead I/O."

All input operations in Pascal are based on obtaining data from a buffer variable. As a result of delayed device access, an item of data is not retrieved from an I/O server and inserted in the buffer variable until the program is ready to process the data. The buffer variable is filled when the program makes the next reference to the file. A reference to the file consists of the use of the buffer variable as a source operand in the GET, READ, and READLN procedures and in the EOF and EOLN functions (for example, variable:= file_variable^).

The RESET procedure, which is required when any file is opened for input, initiates the process of delayed device access. (An OPEN and RESET are done automatically on the predeclared file INPUT.) RESET expects to fill the buffer variable with the first component of the file. However, because of delayed device access, an item of data is not supplied from the input device to fill the buffer variable until the next reference to the file.

When writing a program for which the input data will be supplied by a text file, you should be aware that delayed device access occurs. Since RESET initiates delayed device access and since EOF and EOLN cause the buffer variable to be filled, you should place the first prompt for input from a terminal before any tests for EOF of EOLN. The information you enter in response to the prompt supplies data that is retained by the file device until you make another reference to the input file.

The following example shows the use of prompts in the reading of input data:

Example

```
[SYSTEM (MICROPOWER)] PROGRAM LAZY (INPUT, OUTPUT);
```

```
VAR
   Purch_Amount : REAL;
   .
   .
   .
WRITE
WRITE ('Enter amount of purchase or <CTRL/Z>: ');
WHILE NOT EOF DO
   BEGIN
   READLN (Purch_Amount);
   WRITE ('Enter amount of purchase or <CTRL/Z>: ');
   END;
```

The first reference to the file INPUT is the EOF test in the WHILE statement. When the test is performed, the MicroPower/Pascal run-time system attempts to read a line of input from the text file. So, in this program, you must prompt for the amount of purchase before testing for EOF. If you respond to the prompt by typing CTRL/Z, EOF returns TRUE. If you respond by entering a purchase amount, EOF returns FALSE.

Suppose you respond to the first prompt for input by typing a real number. Access to the input device is delayed until the EOF function makes the first reference to the file INPUT. The EOF function causes a line of text to be read into the internal line buffer. The subsequent READLN procedure reads the input value from the line of text and assigns the input value to the variable Purch_Amount. The final statement in the WHILE loop is the request for another input value. The WHILE loop is executed until EOF detects the end-of-file marker.

A sample run of a program containing this loop might be:

```
$ RUN PURCH
Enter amount of purchase or <CTRL/Z>: 7.95
Enter amount of purchase or <CTRL/Z>: 6.49
Enter amount of purchase or <CTRL/Z>: 19.99
Enter amount of purchase or <CTRL/Z>: ^Z
$
```

The following program fragment shows a method of writing the same loop that does not take into account delayed device access and therefore produces incorrect results:

```
WHILE NOT EOF DO
   BEGIN
   WRITE ('Enter amount of purchase or <CTRL/Z>: ');
   READLN (Purch_Amount);
   END;
```

The EOF test at the beginning of the loop causes the file buffer to be filled. However, because no input has been supplied, the prompt does not appear on the terminal screen until you have supplied input to fill the INPUT file buffer.

A sample run of a program containing this loop might be:

```
$ RUN PURCHASE
7.95
Enter amount of purchase or <CTRL/Z>: 6.49
Enter amount of purchase or <CTRL/Z>: 19.99
Enter amount of purchase or <CTRL/Z>: ^Z
$
```

The prompt always appears after you have typed a value for Purch_Amount.

Chapter 10
Attributes

MicroPower/Pascal attributes increase your control over the properties of variables, routines, processes, and compilation units. Attributes may be specified for type definitions, variable declarations, and PROCEDURE, PROCESS, FUNCTION, PROGRAM, and MODULE headings. In the absence of explicit attributes, Pascal follows default rules to assign properties to variables, routines, processes, and compilation units.

This chapter describes the syntax and meaning of the MicroPower/Pascal attributes. The descriptions are presented in alphabetical order by attribute name. The term "entity" refers to the language element to which an attribute applies.

Appendix F lists the attributes and shows the language entities to which they apply. Table 10–1 summarizes the attributes by functional class.

Table 10–1: MicroPower/Pascal Attributes by Functional Class

Class	Name	Description
Accessibility	READONLY WRITEONLY	Specify how a program can access an entity
Allocation	AT STATIC	Specify the form of storage that an entity will be allocated
Privilege	DEV_ACCESS DRIVER PRIVILEGED	Specify the areas of physical storage that a program (static process) and its subprograms can directly access
Procedure-activation	INITIALIZE TERMINATE	Establish special conditions under which a procedure can be called

Table 10–1 (Cont.): MicroPower/Pascal Attributes by Functional Class

Class	Name	Description
Run-time	CONTEXT DATA_SPACE GROUP INIT_PRIORITY NAME PRIORITY STACK_SIZE SYSTEM	Indicate the default execution-time characteristics of programs (static processes) and dynamic processes
Size	BIT BYTE WORD	Specify the amount of storage to allocate to an entity in a structure
Visibility	EXTERNAL GLOBAL	Allow an entity to be shared between compilation units
Miscellaneous	IDENT	Indicates the program identification or version number
	NOOPTIMIZE	Specifies that optimized code should not be generated for the associated procedure, function, process, or main program
	OPTIMIZE	Specifies that optimized code should be generated for the associated procedure, function, process, or main program
	OVERLAID	Indicates how storage should be allocated for outer-level variables in different compilation units
	POS	Specifies the location of a field in a packed record
	UNSAFE	Disables type checking so an entity can accept values of any type without incurring type-checking errors
	VOLATILE	Specifies that the entity can be subject to unusual side effects during execution

Attributes associated with types usually modify type-compatibility rules. Some attributes, when applied to components of structured types, affect the entire structure. The sections of this chapter pertaining to the READONLY, WRITEONLY, alignment, POS, procedure activation, size, UNSAFE, and VOLATILE attributes describe their effects on type compatibility. The sections discussing the accessibility, size, and volatility attributes also present the rules for using those attributes with structured types. The sections covering run-time environment define the attributes that establish characteristics of the real-time programming environment for programs (static processes) and dynamic processes.

10.1 Specifying Attributes

Attributes become associated with an entity in two explicit ways in a program:

- They can appear in the definition of a user-defined type, and the entity is later declared to be of that type.
- They can appear in the declaration of an entity preceding the type.

If the program does not associate an attribute with an entity, Pascal may automatically supply default attributes for the unspecified entity at the time of the declaration.

A complete list of attributes is associated with an entity of that type only when the type identifier with attributes is used in a declaration.

10.1.1 General Syntax Diagrams

The following diagrams show the syntactic relationship of an attribute and its entity. Complete syntax descriptions of each entity appear in the chapters referenced below.

Syntax

In a type definition (Chapters 2 and 4):

```
TYPE { type-identifier = ⟦ [ {attribute} ,...] ⟧ ⟦PACKED⟧ type ; } ...
```

In a variable declaration (Chapter 4):

```
VAR { {variable-identifier} ,... : ⟦ [ {attribute} ,...] ⟧ ⟦PACKED⟧ type ; } ...
```

In routine and process headings (Chapter 6):

```
⟦ [ {attribute} ,...] ⟧ PROCEDURE procedure-identifier
     ⟦ ( formal-parameter-list ) ⟧ ;

⟦ [ {attribute} ,...] ⟧ FUNCTION function-identifier
     ⟦ ( formal-parameter-list ) ⟧ : result-type-identifier;

⟦ [ {attribute} ,...] ⟧ PROCESS process-identifier
     ⟦ ( formal-parameter-list ) ⟧ ;
```

In formal-parameter lists for routines and processes (Chapter 6):

```
{identifier} ,... : ⟦ [ {attribute} ,...] ⟧ type-identifier ⟦:= default-value⟧

    { {identifier} ,... : ⟦[{attribute} ,...]⟧ type-identifier ⟦:= default-address ⟧ }
VAR {  identifier : conformant-array                                                 }

⟦ [ {attribute} ,...] ⟧
    ARRAY [ {lower-bound-identifier..upper-bound-identifier :
                                              { type-identifier   }
         index-type-identifier} ,...] OF      { conformant-array  }

⟦ [ {attribute} ,...] ⟧
   PACKED ARRAY [lower-bound-identifier..upper-bound-identifier :
          type-identifier] OF CHAR
```

In the heading of a program or module (Chapter 7):

```
⟦ [ {attribute} ,... ] ⟧ { PROGRAM } identifier ⟦ ( {file-variable} ,...) ⟧;
                         { MODULE  }
```

10.1.2 Memory-Mapping Attributes

The memory-mapping attributes specify the areas of physical storage that a program (static process) and its subprograms directly access. These attributes apply only to application environments that use memory-mapping hardware. A mapping type is established for a program when it is declared. Any processes created by the program inherit its mapping type, since the code and data associated with a given program must reside in the same address space. Thus, from the viewpoint of mapping, all processes are "part of" a parent program: One set of address relocation values is used for all processes within a family. Four mapping types can be selected:

- General mapping is for processes that do not require direct access to system data structures, the I/O page, and the interrupt service routines or their data areas. General mapping is the default if no other mapping attribute is specified.
- Device-access mapping allows access to the processor's I/O page and is specified by the DEV_ACCESS attribute.
- Privileged mapping allows access to the processor's I/O page and the kernel's common-data space and is specified by the PRIVILEGED attribute.
- Driver mapping allows access to the processor's I/O page, the kernel's common-data space, and the interrupt service routines and their data areas and is specified by the DRIVER attribute.

General mapping, the standard mapping for most application processes, is intended for processes that do not require direct access to system data structures or to the I/O page. General mapping allows for the largest possible program.

See Chapter 2 of the *MicroPower/Pascal Run-Time Services Manual* for a complete discussion of the MicroPower/Pascal memory-mapping scheme.

10.2 Attribute Descriptions

The remainder of this chapter describes the attributes, which are presented in alphabetical order.

10.2.1 AT

The AT attribute specifies the storage address to allocate for an entity.

Syntax

AT (constant)

constant

A value of type UNSIGNED that specifies a memory address.

Rules and Defaults

- This attribute can be specified in variable declarations.
- A variable with this attribute is assumed to reside at the memory address specified by the constant.
- A variable having the AT attribute is implicitly static.
- Variables representing machine-dependent registers are frequently given the AT attribute.

10.2.2 BIT

The BIT attribute specifies the number of bits of storage to be reserved for a field of a packed record.

Syntax

BIT ⟦ (constant) ⟧

constant
A positive integer that specifies the number of bits to allocate.

Rules and Defaults

- The default allocation size for the entity depends on the data type (see Appendix E).
- The default value for constant is 1.
- The amount of storage described must be large enough to contain an entity of the specified type; otherwise, a compile-time error results.
- Two variables of the same type that have different allocation sizes are assignment compatible.

10.2.3 BYTE

The BYTE attribute specifies the number of bytes of storage to be reserved for a field of a record.

Syntax

BYTE ⟦ (constant) ⟧

constant

A positive integer that specifies the number of bytes to allocate.

Rules and Defaults

- The default allocation size for the entity depends on the data type (see Appendix E).
- The default value for constant is 1.
- The amount of storage described must be large enough to contain an entity of the specified type; otherwise, a compile-time error results.
- Two variables of the same type that have different allocation sizes are assignment compatible.

10.2.4 CONTEXT

The CONTEXT attribute specifies the processor hardware resources used by a program or a process that the kernel must save during a context-switching operation.

Syntax

$$\text{CONTEXT (} \left\{ \begin{array}{l} \text{MMU} \\ \text{NOFPP} \end{array} \right\} \text{,...)}$$

MMU

The kernel is to save and to restore the MMU registers. Use of this option implies that the program or process may be modifying its MMU registers. If not specified, memory mapping is fixed, and the MMU registers are not saved, but they are restored.

NOFPP

The kernel will not save the floating-point processor (FPP) registers. You may select this option to eliminate unnecessary kernel operations for processes that are known not to use the FPP. NOFPP applies only to compilations in which you have selected the compiler's /I:FPP command option. This option is activated by default when you select the /I:NHD, /I:EIS, and /I:FIS compiler command options.

Note

The purpose of the NOFPP option is to permit selective use of the kernel's FPP register save feature. Use extreme care when performing floating-point operations if a program contains processes that disable FPP context switching.

Rules and Defaults

- This attribute can be specified in program and process declarations.
- If MMU is not specified, the memory-mapping hardware should not be modified by the program or process.
- If NOFPP is not specified, the kernel automatically saves and restores the FPP registers if you select the compiler's /I:FPP command string option.

10.2.5 DATA_SPACE

The DATA_SPACE attribute specifies the amount of storage (heap) space in memory to reserve for dynamically allocated program (static process) and dynamic process data.

Syntax

DATA_SPACE (constant)

constant

A positive integer value in the range 0 to 65532 that specifies the number of bytes to allocate.

Rules and Defaults

- This attribute can be specified in program declarations.
- The default value for this attribute is 2000 bytes.
- The compiler allocates data space in multiples of four bytes.
- An exception occurs if a program attempts to use more space than was allocated.

Calculating Data Space

To determine an appropriate data-space value to specify for a program, perform the following steps:

1. For static and dynamic processes:
   a. Obtain the stack size value for each process (see Section 10.2.22).
   b. Add 56 (process impure area size, in bytes) to each value obtained in step 1a.
2. Determine the size, in bytes (see Appendix E), of each dynamic variable allocated by the NEW procedure. Add 3 to that value and round the result down to the next multiple of 4.
3. For each open file:
   a. Calculate the component size (see Appendix E). Add 3 to the value obtained and round the result down to the next multiple of 4. (Opening a file results in a call to NEW to obtain space from the heap.)
   b. Add 76 (file descriptor block size, in bytes) to the value obtained in step 3a.
   c. Obtain the buffer size value specified in the OPEN statement for the file. Add 3 to the value obtained and round the result down to the next multiple of 4. (Buffer space is allocated from the heap through a call to NEW.) If the file is opened with double buffering (OVERLAPPED := ENABLE), multiply this value by 2. Add the value obtained in step 3b to this value.
4. Each CONNECT_SEMAPHORE request requires 16 bytes.
5. Each static or dynamic process that has one or more ESTABLISH requests requires 32 bytes.

6. Using the values obtained in steps 1 through 5, calculate the maximum space necessary to accommodate the largest possible concurrent occurrence of:
   - The static process and dynamic processes
   - Dynamic variables
   - Files
   - CONNECT_SEMAPHORE requests
   - ESTABLISH requests
7. Add the values obtained in step 6.
8. Add 56 (the size, in bytes, of the OTS impure area) to the value obtained in step 7. The result is the minimum data-space value to specify.

Example

The following example demonstrates how to determine the DATA_SPACE value for a simple program:

```
[DATA_SPACE (1136), STACK_SIZE (200)]PROGRAM Test (INPUT, OUTPUT);

  [STACK_SIZE (100)]PROCESS P1;
     BEGIN
        .
        .
        .
     END;

  [STACK_SIZE (150)]PROCESS P2;
    TYPE
      Number_ptr = ^INTEGER;
    VAR
      Number : Number_ptr;

    PROCEDURE EX1;
      BEGIN
      ESTABLISH(...
           .
           .
           .
      END;
```

```
PROCEDURE EX2;
  BEGIN
  ESTABLISH(...
        .
        .
        .
  END;
BEGIN
NEW (Number)
      .
      .
      .
  END;
BEGIN
P1;
P2;
END.
```

Calculate data space requirements as follows:

1. Process stack and impure area size:

```
200+56     static process Test
100+56     dynamic process P1
150+56     dynamic process P2
------
   618 bytes
```

2. Open files:

```
INPUT        4     component size 1 (rounded) = 4
            76     file descriptor
           132     buffer size

OUTPUT       4     component size 1 (rounded)  = 4
            76     file descriptor
           136     66 (rounded) = 68 x 2 for double
           ---     buffering
           428 bytes
```

3. Dynamic variables:

```
   2     integer variable
   2     rounding to multiple of 4
   -
   4 bytes
```

4. Concurrent processes, files, dynamic variables, ESTABLISH requests (assume that all are concurrent):

```
 618     process stack and impure area
 428     open files
   4     dynamic variables
  32     ESTABLISH requests in P2
  56     OTS impure area
----
1138 bytes (absolute minimum value to specify for DATA_SPACE)
```

Application Notes

You can use PASDBG's SHOW HEAP command to dynamically observe allocation from the heap during application execution. Remember that:

- The value returned by SHOW HEAP is not meaningful until the first Pascal statement in your program executes.
- Before the first statement executes, low-level initialization code in the OTS sets up the data space and allocates the process impure area for the static process, the OTS impure area, and the static process stack. The program then runs any initialization procedures declared in the static process.
- After the first statement executes, the value returned by SHOW HEAP is the DATA_SPACE value you specified minus the space required for:
  The static process stack
  The static process impure area
  The OTS impure area
  Files, dynamic variables, ESTABLISH requests, and CONNECT_SEMAPHORE requests resulting from execution of any initialization procedures

10.2.6 DEV_ACCESS

The DEV_ACCESS attribute instructs the compiler to generate object code that allows a program in a mapped-memory environment to access the processor's I/O page (including the device CSRs, MMU registers, and so on) but not the system data structures. DEV_ACCESS mapping is suitable for a program or a process that communicates directly with a dedicated I/O device or for a process that must modify its own mapping. See Section 10.1.2 for general information about memory-mapping attributes and the *MicroPower/Pascal Run-Time Services Manual* for information on programming and virtual address restrictions implied by this attribute.

Syntax

DEV_ACCESS

Rules and Defaults

- This attribute can be specified in program declarations.
- Subprograms that reside in a program with this attribute inherit DEV_ACCESS mapping characteristics.
- The default condition for this attribute is general mapping.

10.2.7 DRIVER

The DRIVER attribute instructs the compiler to generate object code that allows a program in a mapped-memory environment to access directly system data structures (that is, the kernel's common-data space), the I/O page, and the interrupt service routines and their data areas.

DRIVER mapping is suitable for use by device driver processes. See Section 10.1.2 for general information about memory-mapping attributes and the *MicroPower/Pascal Run-Time Services Manual* for information on programming and virtual address restrictions implied by this attribute.

Syntax

DRIVER

Rules and Defaults

- This attribute can be specified in program declarations.
- Subprograms that reside in a program with this attribute inherit DRIVER mapping characteristics.
- The default condition for this attribute is general mapping.

10.2.8 EXTERNAL

The EXTERNAL attribute indicates that an entity is assumed to be defined with the GLOBAL attribute and resident in another compilation unit. Entities with this attribute are not allocated storage space in memory. The data or subprogram body resides in the storage space allocated for the compilation unit that contains the corresponding GLOBAL declaration.

Syntax

EXTERNAL ⟦ (global-id) ⟧

global-id

A 6-character build-time name of the entity. Uppercase and lowercase versions of a character are equivalent. Underscore characters (_) are converted to periods (.).

Rules and Defaults

- This attribute can be specified in declarations for variables, procedures, functions, and processes.
- The EXTERNAL attribute can be specified in declarations for a function, procedure, or process only if that subprogram appears at the outermost level of a program or module. In other words, a subprogram with the EXTERNAL attribute cannot be nested within another subprogram.
- The compiler passes the first six characters of the identifier specified by global-id to the application-build utilities (described in the MicroPower/Pascal system user's guide applicable to your host system). The identifier must be unique within the first six characters across all compilation units. If no identifier is specified for the global-id parameter, the compiler uses the first six characters of the identifier declared for the entity.
- By default, variables and subprograms are not visible to separate compilation units.
- This attribute is illegal on record fields and formal parameters.
- The names and declarations for corresponding GLOBAL and EXTERNAL entities must be identical.
- EXTERNAL subprogram declarations must be followed by a block containing either the EXTERNAL or the SEQ11 directive (see Section 6.5.2).
- A subprogram that is nested (see Section 6.1.3) cannot have the EXTERNAL attribute.

Examples

```
VAR
  Variable_1 : [EXTERNAL] INTEGER;
  Variable_2 : [EXTERNAL] REAL;
```

```
[EXTERNAL (Unique)] PROCEDURE Number_Unique (I:INTEGER);EXTERNAL;
```

Although the variable names above are unique within the compilation unit, both specify the external name Variab. To avoid this conflict, specify a build-time name.

```
VAR
  Variable_1 : [EXTERNAL (V1)] INTEGER;
  Variable_2 : [EXTERNAL (V2)] REAL;
```

In these declarations, the EXTERNAL names for these variables are uniquely named V1 and V2.

10.2.9 GLOBAL

The GLOBAL attribute provides a definition of a variable or a subprogram so other independently compiled units can refer to it through variables or subprograms declared with the EXTERNAL attribute and through subprograms declared with the EXTERNAL or SEQ11 directive (see Section 6.5.2).

Syntax

GLOBAL ⟦ (global-id) ⟧

global-id

A 6-character build-time name of the entity. Uppercase and lowercase versions of a character are equivalent. Underscore characters (_) are converted to periods (.).

Rules and Defaults

- This attribute can be specified in declarations for variables, functions, procedures, and processes.
- The GLOBAL attribute can be specified in declarations for a function, procedure, or process only if that subprogram appears at the outermost level of a program or module. In other words, a subprogram with the GLOBAL attribute cannot be nested within another subprogram.
- The compiler passes the first six characters of the name that is specified by global-id to the application-build utilities (described in the MicroPower/Pascal system user's guide applicable to your host system). The identifier must be unique within the first six characters across all compilation units. If no identifier is specified for the global-id parameter, the compiler uses the first six characters of the identifier declared for the entity.
- By default, variables and subprograms are not visible to separate compilation units.
- This attribute is illegal on record fields and formal parameters.
- The names and declarations for corresponding GLOBAL and EXTERNAL variables and subprograms must be identical.

Example

```
VAR
  Variable_1 : [GLOBAL] INTEGER;
  Variable_2 : [GLOBAL] REAL;
```

Although the variable names above are unique within the compilation unit, both specify the 6-character global name Variab. To avoid this conflict, specify an identifier.

```
VAR
  Variable_1 : [GLOBAL (V1)] INTEGER;
  Variable_2 : [GLOBAL (V2)] REAL;

[ GLOBAL(Unique) ] PROCEDURE UNIQUE_NUMBER(I:INTEGER);
```

In these declarations, the GLOBAL names for these variables are uniquely named V1 and V2.

10.2.10 GROUP

The GROUP attribute specifies that the entity is a member of a specified process exception group. When an exception occurs within a process, the exception handler for this group is called.

Syntax

GROUP (constant)

constant

A positive integer from 1 to 255 that specifies the exception group.

Rules and Defaults

- This attribute can be specified in program declarations and process declarations.
- The default value for this attribute is 1.

10.2.11 IDENT

The IDENT attribute overrides the compiler generated program identification or version number (corresponding to the .IDENT directive in MACRO–11).

Syntax

IDENT (ident-string)

ident-string

A string constant that contains the program identification string. The string must be exactly six characters long. If fewer than six printing characters are used, the string must be padded with trailing spaces. Uppercase and lowercase versions of the same character are treated as unique.

Rules and Defaults

- This attribute can only be specified in program and module declarations.
- If this attribute is not specified, a system generated string is used for the program identification.

Example

```
[IDENT('V2.4b ')]PROGRAM test
BEGIN
   .
   .
   .
END.
```

10.2.12 INIT_PRIORITY

For applications having multiple static processes (programs), the INIT_PRIORITY attribute establishes execution priority for the "INITIALIZE procedures" within the application (see Section 10.2.13). INIT_PRIORITY lets you establish a sequenced startup, in which initialization procedures in different static processes can be executed in a predetermined sequence.

Syntax

INIT_PRIORITY (constant)

constant

An integer value in the range 0 to 255 that specifies the priority value.

Rules and Defaults

- This attribute can be specified in a program declaration.
- This attribute establishes the execution priority of all procedures that have the INITIALIZE attribute.
- The highest priority is 255.
- In the absence of this attribute, the default priority value is 248.
- Within a program, the order of execution of initialization procedures is undetermined.
- Across programs, the order of execution of initialization procedures is determined by their respective priorities.

Example

```
[SYSTEM (MICROPOWER), DATA_SPACE(4000),PRIORITY(15),
INIT_PRIORITY(239)] PROGRAM Routine_Activate;

   [INITIALIZE] PROCEDURE Check_Open;
        .
        .
        .
   BEGIN  (* Routine_Activate *)
        .
        .
        .
   END

[SYSTEM (MICROPOWER), DATA_SPACE(2300),PRIORITY(50),
INIT_PRIORITY(240)] PROGRAM Alarm_Activate;

   [INITIALIZE] PROCEDURE Indicator_Clear;
        .
        .
        .
   BEGIN  (* Alarm_Activate *)
        .
        .
        .
```

These program fragments include initialization procedures among their declarations. At system start-up time, the initialization procedure Indicator_Clear will execute before the initialization procedure Check_Open. Thereafter, the main program Alarm_Activate will execute at priority 50, and main program Check_Open will execute at priority 15.

10.2.13 INITIALIZE

The INITIALIZE attribute specifies that the associated procedure is to be called before control passes to the main part of any program (static process). A procedure with this attribute is called an initialization procedure.

Syntax

INITIALIZE

Rules and Defaults

- This attribute can be specified in procedure declarations.
- A procedure with this attribute must be declared at the outermost level of a program.
- A program can contain any number of procedures with this attribute.
- The default priority value for a procedure with this attribute is 248. You may specify other priorities with the INIT_PRIORITY attribute (see Section 10.2.12).
- Within the same program, the order of execution of procedures with this attribute is not defined.
- Between programs, the order of execution of initialization procedures is undetermined unless the INIT_PRIORITY attribute is used in one or more of the program declarations.
- In the absence of the INITIALIZE attribute, the compiler assumes that a procedure can be activated only by calls within the program.
- A procedure with this attribute cannot use the standard system input/output mechanisms. The procedure can, however, access the I/O page directly.
- A procedure with this attribute can be passed as an actual parameter to a formal routine parameter that does not have this attribute.
- A procedure with this attribute can access only its local variables and variables at the outermost level.
- A procedure with this attribute cannot have a parameter list.
- A procedure with this attribute cannot be declared EXTERNAL.
- A procedure with this attribute can be called as an ordinary procedure.

Example

```
[SYSTEM (MICROPOWER)] PROGRAM Routine_Activate;

   [INITIALIZE] PROCEDURE Check_Open;
      .
      .
      .
   BEGIN  (* Routine_Activate *)
      .
      .
      .
```

This program includes an INITIALIZE procedure among its declarations. The body of Check_Open is executed before the main program is activated.

Applications

A typical task for a procedure with this attribute would be to create global structures. You should not use an initialization procedure to perform tasks that depend on the existence of or refer to structures created by other processes. See Appendix A of the *MicroPower/Pascal Run-Time Services Manual* for a list of suggested user and system process priorities.

10.2.14 NAME

The NAME attribute identifies an invocation of a process.

Syntax

NAME (process-name)

process-name

A string constant or a variable that contains the run-time name of the process. The name must be a 6-character string. If fewer than six printing characters are used, you must pad the name with trailing spaces. Uppercase and lowercase versions of the same character are treated as unique.

Rules and Defaults

- This attribute can be specified in process declarations.
- You may override this specification during process invocation by supplying a value for the predeclared parameter NAME (see Section 5.9).

Example

```
VAR Pname : PACKED ARRAY[1..6] OF CHAR;

[NAME('Proc1 ')] PROCESS Task(I:INTEGER);
   BEGIN
     .
     .
     .
   END;
BEGIN
     .
     .
     .
  Task(I); (* This will have the default name 'Proc1 '. *)

  Pname:='Proc2 ';

  Task(I, NAME:=Pname); (* This will have the name 'Proc2 '. *)
     .
     .
     .
END.
```

10.2.15 NOOPTIMIZE

The NOOPTIMIZE attribute specifies that optimized code should not be generated for the associated procedure, function, process, or main program.

Syntax

NOOPTIMIZE

Rules and Defaults

- This attribute can be specified in procedure, function, process, or program declarations. It has no effect on module declarations.
- The attribute is used to override the effect on optimization by the /DEBUG option; specifying /NODEBUG enables code optimizations for all routines, and the NOOPTIMIZE attribute overrides the optimizations for individual routines.
- If the /DEBUG option is specified, the NOOPTIMIZE attribute has no effect.

Example

```
{compiled with the /NODEBUG option}
VAR b: PACKED ARRAY [1..8] OF BOOLEAN;

{two separate MACRO instructions are generated}
[NOOPTIMIZE]PROCEDURE noopt;
BEGIN
  b[1] := true;
  b[4] := true;
END;

{one optimized instruction is generated}
PROCEDURE opt;
BEGIN
  b[1] := true;
  b[4] := true;
END;
```

10.2.16 OPTIMIZE

The OPTIMIZE attribute specifies that optimized code should be generated for the associated procedure, function, process, or main program.

Syntax

OPTIMIZE

Rules and Defaults

- This attribute can be specified in procedure, function, process, or program declarations. It has no effect on module declarations.
- The attribute is used to override the effect on optimization by the /DEBUG option; specifying /DEBUG suppresses code optimizations for all routines, and the NOOPTIMIZE attribute overrides the suppression for individual routines.
- If the /NODEBUG option is specified, the OPTIMIZE attribute has no effect.

Example

```
{compiled with the /DEBUG option}
VAR b: PACKED ARRAY [1..8] OF BOOLEAN;

{one optimized instruction is generated}
[OPTIMIZE]PROCEDURE opt;
BEGIN
  b[1] := true;
  b[4] := true;
END;

{two separate MACRO instructions are generated}
PROCEDURE noopt;
BEGIN
  b[1] := true;
  b[4] := true;
END;
```

10.2.17 OVERLAID

The OVERLAID attribute allows a program and its modules to share a common outer-level variable storage area. The variables declared at program or module level will, at build time, overlay the storage of outer-level variables in all other compilation units. (Ordinarily, data storage space for each compilation unit is allocated to separate memory areas.)

Syntax

OVERLAID

Rules and Defaults

- This attribute can be specified in program and module declarations.
- By default, variables are not stored in OVERLAID storage regions.
- The VAR, CONST, and TYPE declaration sections of each compilation unit that uses the OVERLAID attribute must be physically and logically identical to ensure proper alignment in physical storage. This identity is easily accomplished by creating a separate file that contains common declarations and inserting them into each program and module with the %INCLUDE directive (see Section 7.3).
- This attribute has no effect on variables defined within procedures, functions, or processes; those variables are allocated storage space dynamically at run time.

10.2.18 POS

The POS attribute specifies the bit position of a field in a packed record.

Syntax

POS (constant)

constant

An integer from 0 to 2*MAXINT+1 that specifies the ordinal bit position, relative to the beginning of the record, at which the field begins.

Rules and Defaults

- Default conditions for the positioning of record fields are described in Appendix E.
- The constant cannot be a negative integer.
- The starting position for a field must be greater than the ending position of the field preceding it.
- Within a record variant declaration, the starting position for a field must be greater than the ending position of the preceding field within the same variant. As always, the variants may overlap.
- A field larger than 16 bits must be positioned on a word boundary.
- The specified bit position must not conflict with the alignment required explicitly by the item.

10.2.19 PRIORITY

The PRIORITY attribute establishes the execution precedence among programs and processes.

Syntax

PRIORITY (constant)

constant
An integer value in the range 0 to 254 that specifies the priority value.

Rules and Defaults

- This attribute can be specified in program and process declarations.
- The highest priority is 255. You may specify this priority only by using the predeclared procedure CHANGE_PRIORITY (see Section 12.1).
- The default priority value for programs is 1.
- The default value for a process is the priority of the program or process that invoked it.
- You can override this specification during process invocation by supplying a value for the predeclared parameter PRIORITY (see Section 5.9).
- You can override this specification during process execution by using the CHANGE_PRIORITY procedure (see Section 12.1).

10.2.20 PRIVILEGED

The PRIVILEGED attribute instructs the compiler to generate object code that will allow a program in a mapped-memory environment to access directly both the system data structures in kernel data space and the I/O page. See Section 10.1.2 for general information about memory-mapping attributes and the *MicroPower/Pascal Run-Time Services Manual* for information on programming and virtual address restrictions implied by this attribute.

Syntax

PRIVILEGED

Rules and Defaults

- This attribute can be specified in program declarations.
- Subprograms that reside in a program with this attribute inherit the mapping characteristics of this attribute.
- The default condition for this attribute is general mapping.

PRIVILEGED mapping is used by exception-handling processes, which generally require access to process control blocks (PCBs). Although PRIVILEGED mapping restricts process size, it allows for efficient queue semaphore operations for interprocess message transmission.

10.2.21 READONLY

The READONLY attribute specifies that an entity can be the source of an assignment statement but not the destination of an assignment statement.

Syntax

READONLY

Rules and Defaults

- This attribute can be specified in declarations for variables, formal parameters, record fields, and type declarations.
- By default, entities can be both read and written.
- You cannot specify more than one accessibility attribute in the same attribute list. That is, an entity cannot be both READONLY and WRITEONLY.
- No value of any type is assignment-compatible with a READONLY entity.
- A READONLY field in a record prohibits the entire record from having values assigned to it. The record as a whole may not be explicitly declared WRITEONLY.
- A READONLY expression used as an actual parameter can be passed only to a READONLY formal parameter.

Example

This example shows a compilation listing that contains both legal and illegal references to a READONLY data item. In Tt_Csr, the field Int_Enb is a read/write item specifying the interrupt enable field of a terminal status register. Done is a read-only field indicating that a character is ready to be processed. Done is under control of the hardware and may not be modified by the user program.

```
              [SYSTEM (MICROPOWER)] PROGRAM R_Only;

                 TYPE
                   Tt_Csr =
                     PACKED RECORD
                       Int_Enb: [POS(6)] BOOLEAN;
                       Done: [POS(7), READONLY, VOLATILE] BOOLEAN;
                     END;

                 VAR
                   Rcsr: [AT(%O'177560')] Tt_Csr;
                   Rchr: [AT(%O'177562')] CHAR;

                 BEGIN

                   (* Initialize the register. *)
                   Rcsr.Int_Enb := FALSE;
                   Rcsr.Done := FALSE;
                         ^171
*** 171: Illegal reference to a readonly data item
```

```

  (* Wait until a character is typed. *)
  WHILE NOT Rcsr.Done DO;

  (* Process the character using Rchr ... *)
END.
```

10.2.22 STACK_SIZE

The STACK_SIZE attribute specifies the amount of stack space used for program and process stacks. This space is allocated from the storage space declared by the DATA_SPACE attribute.

Syntax

STACK_SIZE (constant)

constant
A positive integer value from 0 to 65532 that specifies the number of bytes to allocate.

Rules and Defaults

- This attribute can be specified in program declarations and process declarations.
- The value selected must be a multiple of 4 and be less than that specified for the DATA_SPACE attribute.
- The default value for this attribute is 400 bytes for programs and processes.
- You can override this specification during process invocation by supplying a value for the predeclared parameter STACK_SIZE (see Section 5.9).
- An exception occurs if a program or process attempts to use more space than was allocated for the stack.
- An exception occurs if more space is allocated for the stack than is available.

Calculating Stack Size

To determine an appropriate stack size value to specify for a program or process, perform the following steps:

1. Compile the program or module, using the /E option (see the applicable MicroPower/Pascal system user's guide).
2. Note the stack-depth value for the program or process specified in the compilation listing.
3. For each recursive procedure or function, multiply the stack-depth value by the number of recursions.
4. Add the products obtained in step 3 to the stack-depth value and 56 (OTS work area size).
5. If the process is to reside in an unmapped-memory environment, add 54 (stack space used by the kernel) to the value obtained in step 4 to obtain the stack-size value. Otherwise, in a mapped-memory environment, the value obtained in step 4 is the stack size to use.

10.2.23 STATIC

The STATIC attribute specifies that the storage allocation for a variable be allocated only once. A STATIC variable exists as long as the memory image in which it was allocated remains active.

Syntax

STATIC

Rules and Defaults

- This attribute can be specified in variable declarations and type declarations.
- A variable having the AT attribute is implicitly STATIC.
- Variables having the GLOBAL and EXTERNAL attributes are implicitly STATIC.
- Variables declared with this attribute that reside in a program or a module with the OVERLAID attribute are not allocated from the same storage region as are variables not declared with a storage allocation attribute.
- By default, variables declared at the outer level of a program or module are STATIC.

Note

Statically allocated variables can affect the results of routines that are called recursively.

Example

```
[SYSTEM(MICROPOWER)] PROGRAM Print_Random(OUTPUT);
   FUNCTION Random(New_Seed: INTEGER := 0): INTEGER;
     CONST
       A = 13077;
       C = 6925;
       M = 32768;
     VAR
       Seed: [STATIC] INTEGER;
     BEGIN
       IF New_Seed <> 0
         THEN Seed := New_Seed;
       Seed := ((Seed * A) + C) MOD M;
       Random := Seed;
     END (* Random *) ;
   VAR
     I: INTEGER;
   BEGIN
     (* Provide a new seed for Random. *)
     I := Random(15);
     FOR I := 1 TO 20 DO WRITELN(Random);
   END.
```

The program'Print_Random includes a function that generates a random integer. The variable Seed declared inside the function Random is given the STATIC attribute. Thus, the variable's value will be preserved from one activation of the function to the next. The STATIC attribute overrides the default allocation for inner-level variables, which would have caused the storage for Seed to be deallocated when control returned to the main program after execution of the function. Because Seed is declared STATIC, it retains the value it had when Random ended and uses this value the next time Random is called.

10.2.24 SYSTEM

The SYSTEM attribute selects the run-time environment for a program. This attribute instructs the compiler to generate object code for the specified run-time environment.

Syntax

SYSTEM (environment-name)

environment-name
The identifier MICROPOWER.

Rules and Defaults

- This attribute can be specified in program declarations.
- This attribute is optional and is for documentation purposes only. SYSTEM is intended for use in future versions of this product.
- If you specify this attribute, you must specify MICROPOWER as the environment name.

10.2.25 TERMINATE

The TERMINATE attribute specifies that a procedure will be called by the kernel prior to termination of the program or process in which the procedure resides. The kernel passes control to a termination procedure in response to a STOP request (see Section 12.7), an exception condition (see Chapter 17), or normal process termination.

Syntax

TERMINATE

Rules and Defaults

- This attribute can be specified in procedure declarations.
- A termination procedure must be declared at the outermost level of a program or a process.
- No more than one procedure with this attribute can be declared for a program or a process.
- A termination procedure can also be called as an ordinary procedure.
- In the absence of this attribute, the compiler assumes that a procedure can be activated only by calls within the program.
- A termination procedure cannot have a parameter list.
- A termination procedure cannot be declared EXTERNAL.

The purpose of a termination procedure is to perform operations necessary to ensure an orderly, controlled response to the occurrence of a terminating condition. A typical task for a termination procedure would be to supervise the destruction of data structures created by the subject process or program so the space they occupy can be restored to the heap and to the kernel pool.

Example

```
[SYSTEM (MICROPOWER)] PROGRAM Routine_Activate;
      .
      .
      .
   PROCESS Parallel;
   VAR
     S: SEMAPHORE_DESC;
     Result : BOOLEAN;

   [TERMINATE] PROCEDURE Abort;
      BEGIN (* procedure abort *)
        DESTROY(desc:=S);
      END; (* of procedure abort *)

   BEGIN
    Result := CREATE_BINARY_SEMAPHORE(DESC:=S, NAME:='SEM1  ');
   END;(* Of process parallel *)
```

10.2.26 UNSAFE

The UNSAFE attribute inhibits type checking so an entity can accept values of any type.

Syntax

UNSAFE

Rules and Defaults

- This attribute can be specified in declarations for variables, record fields, type declarations, and formal parameters, except conformant arrays.
- Any expression is assignment compatible with an UNSAFE variable. If the machine representations of the expression's value and the UNSAFE variable differ, the compiler forces them to have the same number of bits by modifying the value of the expression, as follows:
    1. If the UNSAFE variable is larger than the expression's value, assignment is performed as follows:
        a. An expression value smaller than 16 bits is assigned to the low-order bits of the variable. The unused region up to the variable's first word boundary (or byte boundary if the variable is byte sized) is zero filled. The remaining high-order part of the variable (if it exists) is unchanged.

        b. An expression value of 16 bits or larger is assigned to the low-order bits of the variable. If the top of the assigned region falls on other than a byte boundary, the region up to the next byte boundary is zero filled. The remaining high-order part of the variable is unchanged.
    2. If the UNSAFE variable is smaller than the expression's value, the compiler generates a diagnostic error.
- A pointer expression is assignment compatible with an UNSAFE pointer variable if the machine representations of the base types are the same size and if their volatility, accessibility, and alignment attributes allow them to be assignment-compatible.
- An actual value parameter variable may be passed to an UNSAFE formal parameter if the machine representations for both types are the same size and if their volatility, accessibility, and alignment attributes allow them to be structurally compatible.
- An actual VAR parameter may be passed to an UNSAFE formal parameter without regard to its allocation size. However, your application must handle the size difference.

Example

The procedure in this program uses an [UNSAFE] procedure parameter to write nonhomogeneous data into a file. The parameter Buffer is used to pass the data to be written. Because Buffer is declared [UNSAFE], type checking is disabled for the corresponding actual parameter at the procedure call. Thus, any buffer may be passed to Put_Data for output to file. Byte_Count is an integer parameter that informs the procedure how many bytes to write from the specified buffer.

The prócedure Put_Data is called several times in this program to illustrate the many different actual parameter types. The actual parameter need not be an array.

The data buffers in this example are passed by VAR rather than by value to avoid the size-matching restriction on UNSAFE value parameters.

```
[SYSTEM(MICROPOWER)]PROGRAM Data_Out;
  TYPE
    Big = PACKED ARRAY [1..1000] OF CHAR;
  VAR
    Data_File: FILE OF CHAR;
  PROCEDURE Put_Data(VAR Buffer: [UNSAFE,READONLY] Big;
                         Byte_Count: INTEGER);
    VAR
      I: INTEGER;
    BEGIN
      FOR I := 1 TO Byte_Count DO
      BEGIN
        Data_File^ := Buffer[I];
        PUT(Data_File);
      END;
    END (* Put_Data *) ;
  VAR
    I: INTEGER;
    Real_Buf: ARRAY [1..100] OF REAL;
    Int_Buff: ARRAY [1..100] OF INTEGER;
    Chr_Buff: PACKED ARRAY [1..6] OF CHAR;
    Iv: INTEGER;
    Rv: REAL;
  BEGIN

    (* Initialize the data. *)
    Chr_Buff := 'ABCDEF';
    FOR I := 1 TO 100 DO Int_Buff[I] := I;
    FOR I := 1 TO 100 DO Real_Buf[I] := I;
    Iv := 1023;
    Rv := 100.345;

    (* Create the file. *)
    OPEN(Data_File, 'TEMP.DAT', NEW);
    REWRITE(Data_File);

    (* Output the data. *)
    Put_Data(Chr_Buff, SIZE(Chr_Buff));
    Put_Data(Int_Buff, SIZE(Int_Buff));
    Put_Data(Real_Buf, SIZE(Real_Buf));
    Put_Data(Iv, SIZE(INTEGER));
    Put_Data(Rv, SIZE(REAL));

    (* Finish off the file. *)
    CLOSE(Data_File);
  END.
```

10.2.27 VOLATILE

The VOLATILE attribute informs the compiler that the entity may be subject to unusual side effects during execution. Ordinarily, the compiler assumes that an entity will not be subject to unusual side effects.

A VOLATILE entity may change not only in the usual ways but also as the result of an action not directly specified in the program. Thus, the compiler assumes that the value of a VOLATILE entity can be changed or evaluated at any time during program execution. Consequently, a VOLATILE entity does not participate in any optimization based on assumptions about its value. Examples of VOLATILE behavior are modification by asynchronous processes and exception handlers and the behavior of device registers.

Syntax

VOLATILE

Rules and Defaults

- This attribute can be specified in declarations for formal parameters, record fields, type declarations and variables.
- By default, entities are not VOLATILE.
- A variable of a structured type having a VOLATILE component is VOLATILE as a whole. However, the presence of a VOLATILE component does not make other components of the same type VOLATILE.

Example

The following code fragment shows the use of the VOLATILE attribute in declaring a device control/status register (CSR):

```
VAR
  Newchar : CHAR;
  Receiver : [ AT(%O'176500'),
             VOLATILE ] PACKED RECORD
                        Inten : [ POS(6) ] BOOLEAN;
                        Done : [ POS(7), READONLY ] BOOLEAN;
                        Data : [ POS(16), READONLY ] CHAR;
                        Err : [ POS(31), READONLY ] BOOLEAN;
                        END;
BEGIN
  WITH Receiver DO
    BEGIN
      WHILE NOT Done DO;
      IF NOT Err
      THEN Newchar := Data;
    END;
END.
```

Application Note

Multiple processes in a MicroPower/Pascal application can share a variable with access to the shared variable synchronized by a SEMAPHORE or MUTEX structure. However, failure to use the VOLATILE attribute when declaring the variable can produce a form of race condition. The compiler allocates registers for frequently used variables in a procedure, function, or process. If the compiler is allowed to allocate a register for a variable shared with another process, changes to the shared variable might occur in registers rather than in the actual memory location. Here the processes are not using a common value.

Another complication is that during compilation for debugging, the compiler disables the optimization to let you examine the shared variable with PASDBG and obtain correct results. Thus, a program that works correctly during debugging may fail when built for production.

Use of the VOLATILE attribute when declaring a shared variable prevents the compiler from optimizing the value of the variable into a register.

10.2.28 WORD

The WORD attribute specifies the number of words of storage to be reserved for a record field (one word contains two bytes).

Syntax

WORD ⟦ (constant) ⟧

constant

A positive integer that specifies the number of words to allocate.

Rules and Defaults

- The default allocation size for the entity depends on the data type (see Appendix E).
- The default value for constant is 1.
- The amount of storage described must be large enough to contain an entity of the specified type; otherwise, a compile-time error results.
- Two variables of the same type that have different allocation sizes are assignment compatible.

10.2.29 WRITEONLY

The WRITEONLY attribute specifies that the entity can be the destination of an assignment statement but not the source of an assignment statement. The WRITEONLY attribute is used primarily in I/O device register declarations.

Syntax

WRITEONLY

Rules and Defaults

- This attribute can be specified in declarations for formal parameters, record fields, type declarations, and variables.
- By default, entities can be both read and written.
- A WRITEONLY entity cannot be combined in expressions with other values of the same type.
- A WRITEONLY component of a record prohibits the entire record from being read by a program. The record type as a whole may not be explicitly declared READONLY.
- A WRITEONLY expression used as an actual parameter can be passed only to a WRITEONLY formal parameter.
- A pointer to a WRITEONLY base type is not assignment compatible with a pointer to a base type that is not WRITEONLY.

Caution

The size of a WRITEONLY entity should be one word only. Otherwise, the compiler generates object code that may cause some models of the target processor to perform a read-modify-write memory cycle and thus read the entity.

Example

This example shows a compilation listing of a program that contains some legal and illegal references to a write-only data item. The character buffer to the serial interface, Xchr, is a write-only field and may not be examined by the program. The character buffer may, however, be loaded from the program.

```
         1        [SYSTEM (MICROPOWER)] PROGRAM W_Only;
         2
         3           TYPE
         4             Tt_Csr =
         5               PACKED RECORD
         6                 Int_Enb: [POS(6)] BOOLEAN;
         7                 Done: [POS(7), READONLY, VOLATILE] BOOLEAN;
         8               END;
         9
        10           VAR
        11             Xcsr: [AT(%O'177564')] Tt_Csr;
        12             Xchr: [AT(%O'177566'), WRITEONLY] CHAR;
        13
        14
        15           BEGIN
        16
        17             (* Initialize the register. *)
        18             Xcsr.Int_Enb := FALSE;
        19
        20             (* Put out a char. *)
        21             Xchr := 'A';
        22             (* Wait until the interface is done. *)
        23             WHILE NOT Xcsr.Done DO;
        24
        25             (* Continue processing .... *)
        26             IF Xchr = 'A'
                           ^169
*** 169: Illegal reference to a writeonly data item
        27                 THEN;
        28           END.
```

Chapter 11

Introduction to Real-Time Programming Requests

This chapter introduces the real-time programming requests that are the Pascal language interface to the primitive services of the MicroPower/Pascal kernel (see *MicroPower/Pascal Run-Time Services Manual,* Chapter 3). The requests consist of a set of procedure and function calls that are predeclared in one of several system files that you can include in your program or module with the %INCLUDE directive. Appendix I lists those files and the requests they define.

Table 11–1 lists the real-time programming requests by functional groups.

Table 11–1: MicroPower/Pascal Real-Time Programming Requests by Functional Group

Clock Services		
	GET_TIME	SLEEP
	SET_TIME	
Descriptor Initialization		
	INIT_PROCESS_DESC	INIT_STRUCTURE_DESC
Exception Condition Control		
	CONNECT_EXCEPTION	REPORT
	DISCONNECT_EXCEPTION	REVERT
	ESTABLISH	WAIT_EXCEPTION
	RELEASE_EXCEPTION	

Table 11-1 (Cont.): MicroPower/Pascal Real-Time Programming Requests by Functional Group

I/O Control		
	CONNECT_INTERRUPT	DISCONNECT_INTERRUPT
	CONNECT_SEMAPHORE	DISCONNECT_SEMAPHORE
Memory Allocation and Mapping		
	ACCESS_SHARED_REGION	MAP_WINDOW
	ALLOCATE_REGION	RESTORE_CONTEXT
	CREATE_SHARED_REGION	SAVE_CONTEXT
	DEALLOCATE_REGION	UNMAP_WINDOW
	DELETE_SHARED_REGION	
Miscellaneous		
	CREATE_LOGICAL_NAME	POWER_FAIL
	DELETE_LOGICAL_NAME	TRANSLATE_LOGICAL_NAME
	GET_CONFIG	
Process Management		
	CHANGE_PRIORITY	SCHEDULE
	GET_STATE	STOP
	RESUME	SUSPEND
Process Message Transmission		
	COND_GET_PACKET	GET_PACKET_ANY
	COND_PUT_PACKET	PUT_PACKET
	COND_RECEIVE	RECEIVE
	COND_RECEIVE_ACK	RECEIVE_ACK
	COND_SEND	RECEIVE_ANY
	COND_SEND_ACK	SEND
	GET_PACKET	SEND_ACK

Table 11–1 (Cont.): MicroPower/Pascal Real-Time Programming Requests by Functional Group

Process Synchronization	
COND_SIGNAL	SIGNAL
COND_WAIT	SIGNAL_ALL
DEFINE_STOP_FLAG	UNLOCK_MUTEX
GET_VALUE	WAIT
LOCK_MUTEX	WAIT_ANY
Resource Allocation/Deallocation	
ALLOCATE_PACKET	CREATE_QUEUE_SEMAPHORE_P
CREATE_BINARY_SEMAPHORE	CREATE_RING_BUFFER
CREATE_BINARY_SEMAPHORE_P	CREATE_RING_BUFFER_P
CREATE_COUNTING_SEMAPHORE	DEALLOCATE_PACKET
CREATE_COUNTING_SEMAPHORE_P	DESTROY
CREATE_MUTEX	DESTROY_MUTEX
CREATE_QUEUE_SEMAPHORE	
Ring Buffer Management	
COND_GET_ELEMENT	GET_ELEMENT_ANY
COND_PUT_ELEMENT	PUT_ELEMENT
GET_ELEMENT	RESET_RING_BUFFER

11.1 General Conventions and Usage Rules

The real-time programming requests are supplied as standard Pascal procedures and functions. The general rules of usage described in Chapter 6 apply, except that you should use only the nonpositional parameter (keyword) syntax when passing actual parameters to a request. Do not use the positional parameter syntax; unpredictable results may occur. Not enough information about the formal parameter definitions is supplied to allow you to specify the correct positional syntax.

When you specify more than one parameter, use a comma to separate each parameter from the next.

11.1.1 Name and Descriptor Parameters

Most of the real-time programming requests have parameters that include the specification of a name or a descriptor or both. Both names and descriptors identify a process or a structure such as a semaphore or a ring buffer, for example.

A name is a system-wide identifier for a structure or for a particular invocation of a process. The name provides access to a structure or a process across the boundaries of a mapped-memory environment. Names for structures and processes must be unique among themselves throughout the MicroPower/Pascal application environment.

A descriptor is a variable initialized to contain an identifier of a structure or a process. You allocate and initialize descriptors in process space. In subsequent uses of the initialized descriptor, the identifier permits direct, optimized access to the structure or process. Since the table-lookup step performed by the kernel for a reference by name is bypassed, the request is processed more quickly.

11.1.1.1 Specifying Names

A name is a 6-character ASCII string that globally names a structure or a process. If you use fewer than six printing characters, you must pad the name with trailing spaces. Uppercase and lowercase characters are interpreted uniquely, unlike characters in Pascal identifiers and keywords.

You should not define a name that contains a dollar sign ($), which is reserved for use by DIGITAL-supplied software. You may, of course, specify a name that contains a dollar sign when the name refers to a DIGITAL-supplied software component.

You can create a logical equivalent for a name with the logical name requests described in Chapter 20.

11.1.1.2 Specifying Descriptors

Before you use a descriptor, it must be declared as a variable of a predefined descriptor type and be initialized by one of the CREATE or INIT requests, as applicable. Those requests copy a structure or a process identifier into a variable.

11.1.1.3 Using Descriptors for Unnamed Structures

If a structure is not named, the structure descriptor block (SDB) provides the only access path to it. To refer to an existing unnamed structure, the calling process must supply a descriptor variable that contains a valid structure identifier. Thus, in order to access such a structure, a process other than the creator must also have access to the descriptor variable used to create it. In an unmapped memory environment, the descriptor variable of an unnamed structure would be easily accessible, within Pascal scoping rules, to all processes. In a mapped environment, however, an unnamed structure's descriptor variable can be shared only among processes residing in the same address space, for most practical purposes. (However, the descriptor could be sent as a message to another process.)

11.1.1.4 Process Descriptor Usage

A process descriptor identifies a process control block (PCB), the system structure that stores the current state and context of an existing process. Like a structure descriptor, a process descriptor provides efficient access to a process when using the process management requests.

The rules for reference to an existing process differ from those for reference to an existing structure in that the descriptor parameter to any of those requests has a default value—the identity of the calling process. Requests that operate on existing processes thus allow the calling process to specify itself as the process to be acted on—as well as some other process.

11.1.2 STATUS Parameter

The optional STATUS parameter is provided for those predeclared requests for which recovery from a run-time error may be possible. Thus, a process can handle request-related errors locally.

When you specify this parameter, errors relating to the request will be intercepted by the Pascal object-time system (OTS) rather than being reported as exceptions. Each time a request is issued, the OTS copies a status record into the variable specified by the parameter to indicate the results, success or error, of the request; the process execution state is not altered. In the case of an error, the status record contains the exception type and code associated with the error condition. In the case of no error (success), the status record's code field will contain the code ES$NOR. The routine issuing the request can examine the status record to determine the appropriate action. If you do not specify the STATUS parameter, the OTS reports the error as an exception, which is handled by an exception-handling process or procedure or causes the offending process to be aborted.

The variable that you specify for the STATUS parameter must be of predefined type EXC_STATUS, for example:

```
EXC_STATUS = RECORD
                EXC_TYPE : EXC_SET;
                EXC_CODE : UNSIGNED;
              END;
```

EXC_TYPE
A variable of predefined type EXC_SET that indicates the exception type (see Table 17–1).

EXC_CODE
A variable of predefined type UNSIGNED that indicates the exception code (see Table 17–1).

11.2 Error Returns from Real-Time Programming Requests

The MicroPower/Pascal kernel recognizes an error condition caused by the execution of a real-time programming request. Those errors may cause an exception condition depending on the error-handling policy of your application (see Chapter 17).

The Error Returns section in the description of each request lists the errors that can cause exceptions. Some of the errors listed are described as "not occurring when using standard Pascal programming practice." The errors occur when an invalid parameter is passed through a real-time request to a kernel primitive, which is usually the result of careless use of one of the "nonstandard" MicroPower/Pascal features on a parameter.

Typical examples are:

- Failure to initialize a variable—An uninitialized descriptor variable is passed as an actual parameter.
- A parameter is type cast.
- A parameter is defined as a case variant, and the wrong case variant is selected.
- A global variable is used that has incompatible GLOBAL/EXTERNAL type declarations.

The *MicroPower/Pascal Run-Time Services Manual* contains detailed information about exception conditions. Chapter 3 describes the exception management primitives, and Chapter 7 describes the kernel's exception condition management strategy. Chapter 17 of this manual lists all exceptions and describes the requests that let you create exception-handling processes and procedures.

Chapter 12
Process Management Requests

This chapter describes the requests that provide for process execution control. These requests, implemented through the predeclared procedures and functions listed in Table 12–1, are the Pascal language interface to the services provided by the kernel's process management primitives. The requests provide for process access, control, forced termination, resumption, status reporting, and suspension. Chapter 2 of the *MicroPower/Pascal Run-Time Services Manual* provides more detail on process concepts.

You must include in your program or module the definitions of the routines that implement the requests described in this chapter before using them. See Appendix I for more information.

Table 12–1 summarizes the operations performed by these requests.

Table 12–1: Operations Performed by Process Management Requests

Request	Operation
CHANGE_PRIORITY	Changes a process's priority
DEFINE_STOP_FLAG	Permits a process to disable a STOP request issued against it
GET_STATE	Obtains information about a process
INIT_PROCESS_DESC	Sets up a descriptor for efficient reference to a process
RESUME	Reactivates a suspended process
SCHEDULE	Relinquishes the CPU to another process having the same priority
STOP	Stops a process
SUSPEND	Suspends an active process

12.1 CHANGE_PRIORITY

MACRO equivalent: CHGP$

The CHANGE_PRIORITY procedure changes the execution priority of a process to the value specified in the call. Thus, the calling process can dynamically modify its own scheduling priority or that of another process.

Typically, a process uses this procedure to lower its priority to a normal operating level (less than 248) after starting at a high priority level, as may be required for initialization tasks other than those with the INITIALIZE attribute (see Chapter 10). The special start-up priorities are from 248 to 255; the highest is 255. See the *MicroPower/Pascal Run-Time Services Manual* for additional information about priorities.

Procedures with the INITIALIZE attribute (see Chapter 10) automatically execute at a default priority of 248. When execution is complete, the run-time system automatically lowers the priority to the value you specified in the program's heading. The 1-time initialization sequences involving creation of global system structures, and possibly subprocesses, can thus occur before processes that use those resources begin operating. Other processes may use start-up priorities from 248 to 254 to guarantee a particular starting order among a group of related processes.

In general, you should create system-wide resources at a priority level higher than any normal operating priority used in the system. That prevents start-up race conditions among processes in different process families.

Syntax

```
CHANGE_PRIORITY ( PRIORITY := process-priority
                  [[ { DESC := process-descriptor } ]]
                  [[ { NAME := process-name       } ]]
                  [[ STATUS := status-record ]] )
```

process-priority

The identifier of a variable of predefined type PRIORITY_RANGE that specifies the new priority of the process. This value must be from 0 to 255.

process-descriptor

The identifier of a variable of predefined type PROCESS_DESC that contains the process identifier. The variable must have been previously initialized by an INIT_PROCESS_DESC request or by a process invocation statement with the DESC parameter.

process-name

Either a character-string constant or the identifier of a variable of predefined type NAME_STR that contains the 6-character name of an existing process (see Section 11.1.2).

status-record

The identifier of a variable of predefined record type EXC_STATUS that may receive an exception type and code. If you specify this parameter, the exception status, either success or error, that results from issuing the request is reported in this variable. Otherwise, an error will cause the corresponding exception to be reported. The format of the exception record is described in Section 11.1.2.

If you do not specify either a process-descriptor or a process-name parameter, the CHANGE_PRIORITY procedure changes the execution priority of the process issuing the request.

Example

```
[PRIORITY(10), STACK_SIZE(100), NAME ('DRIVER')] PROCESS Driver;
BEGIN

  (* Change the priority of this process. *)
  CHANGE_PRIORITY (PRIORITY := 3);

END; (* Process Driver *)
```

Semantics

The CHANGE_PRIORITY procedure requests the kernel to place the specified priority value in the process control block of the identified process and to call the scheduler. If no process is explicitly identified in the call, the request alters the priority of the calling process, causing the calling process to be preempted if a process with a priority higher than the new priority is in the ready-active state at the time of the call. Otherwise, control returns to the calling process.

This request is implemented through the CHGP$ kernel primitive.

Error Returns

See Section 11.2 for general information about error returns. The following exception code may be returned:

ES$IST	(type: SYSTEM_SERVICE)—Illegal structure descriptor; no such process exists

The request may return the following error, though not as a result of standard Pascal programming practice:

ES$IPM	(type: SYSTEM_SERVICE)—Illegal parameter; the specified priority is less than 0 or greater than 255

12.2 DEFINE_STOP_FLAG

MACRO equivalent: SSFA$

The DEFINE_STOP_FLAG procedure lets the calling process disable the effect of a STOP (process) request issued against the caller by some other process and to receive instead an indication that such a request has occurred. More specifically, DEFINE_STOP_FLAG establishes a Boolean stop-flag variable, which the kernel sets to TRUE if and when another process issues a STOP request against the subject process.

DEFINE_STOP_FLAG also allows the caller to eliminate a previously established stop flag, which effectively reenables the normal, immediate effect of a STOP request issued against the caller. Note that the existence of a stop flag for a given process does not inhibit the process from stopping itself with a reflexive call to STOP nor does it inhibit an implicit stop (or process abort) resulting from an unhandled exception condition.

Syntax

DEFINE_STOP_FLAG ⟦ (FLAG := stop-flag) ⟧

stop-flag

The identifier of a variable of type BOOLEAN that the kernel will use as a stop flag for the calling process. You must initialize this variable to FALSE prior to invoking the request. Not specifying this parameter requests that use of the caller's existing stop flag be discontinued.

Restriction

The kernel assumes that the specified Boolean variable is FALSE when the request is issued.

Example

```
%INCLUDE 'MISC.PAS'

PROCESS A;
VAR
  Stop_flag : BOOLEAN;

BEGIN

  (* Indicate that we do not want to be stopped. *)
  Stop_flag := FALSE;
  DEFINE_STOP_FLAG (FLAG := Stop_flag);

END; (* Process A *)
```

Semantics

The DEFINE_STOP_FLAG procedure instructs the kernel to inhibit STOP requests issued against the caller and instead to set the value of a specified Boolean variable to TRUE. If no variable is specified, use of the caller's current stop flag is discontinued. This null operation occurs if there is no currently defined stop flag.

This request is implemented through the SSFA$ kernel primitive.

Error Returns

See Section 11.2 for general information about error returns. The request may return the following errors, though not as a result of standard Pascal programming practice:

ES$IAD	(type: SYSTEM_SERVICE)—Invalid address; the specified stop-flag location is not within the process's address space (mapped only)
ES$IPR	(type: SYSTEM_SERVICE)—Illegal primitive; the request was issued from an ISR

Application Note

DEFINE_STOP_FLAG is intended to permit a process to defer execution of its termination routine, in response to a STOP request, until an appropriate point is reached in its normal execution cycle or to modify its normal execution path before terminating. The subject process can periodically test its stop flag (for example, just before issuing an I/O request) and take appropriate action depending upon the TRUE or FALSE state of the flag. To further the example, if the flag value were TRUE the process could gracefully terminate its I/O operations and perform any required signals to other processes before executing a STOP on itself.

12.3 GET_STATE

MACRO equivalent: GTST$

The GET_STATE procedure obtains information about a process. The data includes the state code, group, priority, abort status, and type of a process, as well as its suspension count.

Note

Process information is dynamic and, except for group, priority, and type data, may be invalid by the time it becomes available to the requestor.

Syntax

```
GET_STATE (  STATE := state-record
            [[ { DESC := process-descriptor  } ]]
            [[ { NAME := process-name        } ]]
            [[ STATUS := status-record ]] )
```

state-record

The identifier of a variable of predefined type STATE_BLOCK that receives the process state data.

process-descriptor

The identifier of a variable of predefined type PROCESS_DESC that contains the process identifier. The variable must have been previously initialized by an INIT_PROCESS_DESC request or by a process invocation statement with the DESC parameter.

process-name

Either a character-string constant or the identifier of a variable of predefined type NAME_STR that contains the 6-character name of an existing process (see Section 11.1.2).

status-record

The identifier of a variable of predefined record type EXC_STATUS that may receive an exception type and code. If you specify this parameter, the exception status, either success or error, that results from issuing the request is reported in this variable. Otherwise, an error causes the corresponding exception to be reported. The format of the exception record is described in Section 11.1.2.

If you specify neither a process-descriptor nor a process-name parameter, the procedure obtains the state of the process issuing the request.

State Record Format

The state record is a variable of predefined type STATE_BLOCK as defined in the following:

```
STATE_BLOCK = PACKED RECORD
            PRIORITY : [BYTE] PRIORITY_RANGE;
            STATE : [BYTE] PROCESS_STATE;
            TYP : [BYTE] PROCESS_TYPE;
            STATE_CODE_MODIFIER : [BYTE] STATE_CODE_MODIFIER_TYPE
            GROUP : [BYTE] EXC_GROUP;
            RESERVED : [BYTE] BYTE_RANGE;
            BLOCKING_SEMAPHORE : UNIVERSAL ;
            SUSPEND_COUNT : INTEGER;
          END;
```

PRIORITY

The value of type PRIORITY_RANGE that specifies the priority of the process.

STATE

A variable of enumerated type PROCESS_STATE that indicates the execution state of the process. PROCESS_STATE type is defined as follows:

```
PROCESS_STATE = ( RUN, READY_ACTIVE, READY_SUSPENDED,
                  WAIT_ACTIVE, WAIT_SUSPENDED,
                  EXCEPTION_WAIT_ACTIVE,
                  EXCEPTION_WAIT_SUSPENDED );
```

TYP

A variable of the predefined type PROCESS_TYPE that indicates the process's mapping type as specified in the program heading. PROCESS_TYPE is defined as follows:

```
PROCESS_TYPE = ( GENERAL, PRIVILEGED, DEV_ACCESS, DRIVER )
```

STATE_CODE_MODIFIER

A variable record of the predefined type [BYTE] STATE_CODE_MODIFIER_TYPE that provides additional information about the process state. The record is defined as follows:

```
[BYTE] STATE_CODE_MODIFIER_TYPE = PACKED RECORD
                  RES1, RES2, RES3, RES4 : BOOLEAN;
                  UNBLOCK_IN_PROGRESS : BOOLEAN;
                  ABORT_PENDING : BOOLEAN;
                  ABORTED : BOOLEAN;
                END;
```

RES1, RES2, RES3, RES4

Reserved by DIGITAL.

UNBLOCK_IN_PROGRESS

Used by the kernel for internal operations.

ABORT_PENDING

A Boolean variable; when TRUE, indicates that the process is in one of the exception-wait states and that another process has issued a STOP request for this process.

ABORTED

A Boolean variable; when TRUE, indicates that the process is being aborted.

GROUP

A variable of predefined type EXC_GROUP that indicates the exception-handling group to which this process is assigned. This value is from 0 to 255.

RESERVED

Reserved by DIGITAL.

BLOCKING_SEMAPHORE

A variable of predefined type UNIVERSAL that receives the value of the PC.SPT field of the process control block (PCB) (see Chapter 2 of the *MicroPower/Pascal Run-Time Services Manual*).

SUSPEND_COUNT

An integer variable that contains a count of the number of SUSPEND or RESUME requests issued for the process. A negative sign indicates that the value is the number of SUSPEND requests; a positive sign, that the value is the number of RESUME requests. A value of 0 indicates that the process is not suspended and that no RESUME requests are pending.

Example

```
TYPE
  Lines = 0..15;

VAR
  Line1_state, Line2_state, Line3_state : STATE_BLOCK;
  Line2_desc : PROCESS_DESC;

[PRIORITY(10), STACK_SIZE(100)] PROCESS Control (n : Lines);
BEGIN

  (* Get the state information for this process. *)
  GET_STATE (STATE := Line1_state);

  (* Get the state information for the process whose *)
  (* id is in Line2_desc. *)
  GET_STATE (STATE := Line2_state, DESC := Line2_desc);

  (* Get the state for the process with name 'LINE3 '. *)
  GET_STATE (STATE := Line3_state, NAME := 'LINE3 ');

END; (* Process Control *)
```

Semantics

The GET_STATE procedure requests the kernel to copy state information from the process control block (PCB) of the specified process to a record in the caller's storage area and to return control to the caller.

This request is implemented through the GTST$ kernel primitive.

Error Returns

See Section 11.2 for general information about error returns. The following exception codes may be returned:

ES$CDN	(type: SYSTEM_SERVICE)—Cannot specify both name and descriptor
ES$IST	(type: SYSTEM_SERVICE)—Illegal structure descriptor; no such process exists

The request may return the following error, though not as a result of standard Pascal programming practice:

ES$IAD	(type: SYSTEM_SERVICE)—Invalid address; pointer to buffer or structure is odd or not in user address space

12.4 INIT_PROCESS_DESC

MACRO equivalent: GTST$

The INIT_PROCESS_DESC procedure copies process identification information into a process-descriptor record, which provides the kernel with a rapid-access path to a process referred to in the other process management requests described in this chapter.

Syntax

```
INIT_PROCESS_DESC (  DESC := process-descriptor
                     [[ NAME := process-name ]]
                     [[ STATUS := status-record ]] )
```

process-descriptor

The identifier of a variable of predefined type PROCESS_DESC that will receive the structure identifier of the process.

process-name

A character-string constant or the identifier of a variable of predefined type NAME_STR that contains the 6-character name of an existing process (see Section 11.1.2). If you do not specify a name, the procedure initializes the descriptor of the process issuing the request.

status-record

The identifier of a variable of predefined record type EXC_STATUS that may receive an exception type and code. If you specify this parameter, the exception status, either success or error, that results from issuing the request is reported in this variable. Otherwise, an error causes the corresponding exception to be reported. The format of the exception record is described in Section 11.1.2.

Example

```
VAR
  Reader_desc, Writer_desc : PROCESS_DESC;

[PRIORITY(10), STACK_SIZE(100), NAME ('READER')] PROCESS Reader;
BEGIN

  (* Get the id of this process. *)
  INIT_PROCESS_DESC (DESC := Reader_desc);

  (* Get the id of the process with name 'WRITER'. *)
  INIT_PROCESS_DESC (DESC := Writer_desc, NAME := 'WRITER');

END; (* Process Reader *)
```

Semantics

The INIT_PROCESS_DESC procedure requests the kernel to copy the index and serial number of the specified process into the specified descriptor variable in the caller's local storage.

This request is implemented through the GTST$ kernel primitive.

Error Returns

See Section 11.2 for general information about error returns. The following exception code may be returned:

ES$IST	(type: SYSTEM_SERVICE)—Illegal structure descriptor; no such process exists

The request may return the following error, though not as a result of standard Pascal programming practice:

ES$IAD	(type: SYSTEM_SERVICE)—Invalid address; pointer to buffer or structure is odd or not in user address space

12.5 RESUME

MACRO equivalent: RSUM$

The RESUME function attempts to restore a process to an active state. The function returns a Boolean TRUE or FALSE value to indicate the result of the operation.

Syntax

$$\text{RESUME} ⟦ (\left\{ \begin{array}{l} \text{DESC := process-descriptor} \\ \text{NAME := process-name} \end{array} \right\}$$
$$⟦ \text{STATUS := status-record} ⟧) ⟧$$

process-descriptor
The identifier of a variable of predefined type PROCESS_DESC that contains the process identifier. The variable must have been previously initialized by a process invocation or by an INIT_PROCESS_DESC request.

process-name
Either a character-string constant or the identifier of a variable of predefined type NAME_STR that contains the 6-character name of an existing process (see Section 11.1.2).

status-record
The identifier of a variable of predefined record type EXC_STATUS that may receive an exception type and code. If you specify this parameter, the exception status, either success or error, that results from issuing the request is reported in this variable. Otherwise, an error causes the corresponding exception to be reported. The format of the exception record is described in Section 11.1.2.

If you specify neither a process-descriptor nor a process-name parameter, the procedure resumes the process issuing the request.

Example

```
TYPE
  Lines = 0..15;

VAR
  Line2_desc : PROCESS_DESC;

[PRIORITY(10), STACK_SIZE(100)] PROCESS Control (n : Lines);
BEGIN

  (* Unconditionally increment the suspend count of this process. *)
  IF RESUME
   THEN
   ELSE;

  (* Unconditionally increment the suspend count of the process *)
  (* whose id is in Line2_desc. *)
  IF RESUME (DESC := Line2_desc)
   THEN
   ELSE;
  (* Unconditionally increment the suspend count of the process *)
  (* whose name is 'LINE3 '. *)
  IF RESUME (NAME := 'LINE3 ')
   THEN
   ELSE;

END; (* Process Control *)
```

Semantics

The RESUME function increments the suspension count associated with the specified process. If the count changes from -1 to 0, the state of the process is changed to ready active, wait active, or exception-wait active, depending on its state at the time of resumption. If the new state is ready active and the process is of higher priority than the caller, the process is placed in the run state. Otherwise, control returns to the caller after incrementing the count.

Depending on the value of the suspension count, a RESUME request may not reactivate a process. (For example, if the suspension count was less than -1, the process is still suspended, and no state transition occurs.)

The immediate effect of the request is indicated as a Boolean function return. A TRUE return indicates that the process either was reactivated (changed from the suspended state to the appropriate active state) or was already active. A FALSE return indicates that the process was not changed from the suspended state because the suspension count remained negative after the increment.

The maximum value of the suspension counter is 32,767; that is, the counter can record a maximum of 32,767 successive SUSPEND or RESUME requests.

This request is implemented through the RSUM$ kernel primitive.

Error Returns

See Section 11.2 for general information about error returns. The following exception codes may be returned:

ES$CDN	(type: SYSTEM_SERVICE)—Cannot specify both name and descriptor
ES$IST	(type: SYSTEM_SERVICE)—Illegal structure descriptor; no such process exists

12.6 SCHEDULE

MACRO equivalent: SCHD$

The SCHEDULE procedure blocks the calling process if another process of equal priority is in the ready-active state.

This procedure permits a process to choose when to relinquish the CPU to another process of equal priority, allowing some degree of CPU sharing among such processes.

Syntax

SCHEDULE

Example

```
[PRIORITY(10), STACK_SIZE(100), NAME ('INVERT')] PROCESS Invert;
BEGIN

  (* We have used our share of the CPU, let someone else run. *)
  SCHEDULE;

END; (* Process Invert *)
```

Semantics

If the first process in the ready-active queue has the same priority as the caller, the SCHEDULE procedure places the calling process on the ready-active queue behind all processes of the same priority. The first process in the queue is placed in the run state.

If no process of equal priority is in the ready-active state at the time of the call, control returns immediately to the caller.

This request is implemented through the SCHD$ kernel primitive.

Error Returns

None

12.7 STOP

MACRO equivalent: STPC$

The STOP procedure halts a specified process by scheduling the process for execution at its termination procedure, assuming that the subject process does not have a stop flag in effect. If blocked or suspended at the time of the call to STOP, the process is placed in the ready-active state, as described under Semantics. The process is assigned a special "aborted" status that prevents the process from being subsequently suspended.

The request may return control to the calling process, depending on the relative priorities of the caller and the subject processes. (The calling process and the process being stopped may be one and the same.)

When a stopped process begins execution at its termination procedure, the process must determine what action to take before deleting itself. Minimally, the process should deallocate any resources it owned; for example, it should delete any semaphores or other structures that it created, return any packets to the kernel's free-packet pool, and so forth. Prior to resource deallocation, the process could take any actions needed for a graceful termination, such as completing an in-progress I/O operation or message transmission. At the appropriate point, the process deletes itself from the system by allowing the sequence of statement execution to proceed to the END statement.

Alternatively, if the subject process has a stop flag in effect (see DEFINE_STOP_FLAG) and the subject process is not the caller, STOP simply sets the subject process's stop flag to TRUE and has no further effect.

You define a termination procedure for a process by declaring within the process a procedure having the TERMINATE attribute (see Section 10.2.25). If no termination procedure is defined for the process, it terminates immediately when placed in the run state.

Syntax

```
STOP ⟦ ( { DESC := process-descriptor }
         { NAME := process-name       }
         ⟦ STATUS := status-record ⟧ ) ⟧
```

process-descriptor
The identifier of a variable of predefined type PROCESS_DESC that contains the process identifier. The variable must have been previously initialized by a process invocation or by an INIT_PROCESS_DESC request.

process-name
Either a character-string constant or the identifier of a variable of predefined type NAME_STR that contains the 6-character name of an existing process (see Section 11.1.2).

status-record
The identifier of a variable of predefined record type EXC_STATUS that may receive an exception type and code. If you specify this parameter, the exception status, either success or error, that results from issuing the request is reported in this variable. Otherwise, an error causes the corresponding exception to be reported. The format of the exception record is described in Section 11.1.2.

If you specify neither a process-descriptor nor a process-name parameter, the procedure stops the process issuing the request.

Example

```
VAR
  P2_desc : PROCESS_DESC;

[PRIORITY(10), STACK_SIZE(100), NAME ('P1    ')] PROCESS P1;
BEGIN

  (* Stop this process. *)
  STOP;

  (* Stop the process whose id is in P2_desc. *)
  STOP (DESC := P2_desc);

  (* Stop the process with name 'P3    '. *)
  STOP (NAME := 'P3    ');

END; (* Process P1 *)
```

Semantics

If the subject process does not have a stop flag in effect or if the subject process is the caller, STOP will:

- Modify the context of the subject process so the process's execution will resume at its termination entry point when the process is subsequently scheduled for execution, or when control is returned in the case of a "self stop" request
- Flag the process with a special "aborted" status indication, which prohibits any later SUSPEND or STOP of the subject process

If the subject process is in the ready-active or the run state, control returns to the calling process. If the subject process is the caller, control also returns to the calling process. If the subject process is not in the ready-active or the run state, one of the following cases applies:

- If blocked on a binary, counting, or queue semaphore (wait-active state), the subject process is removed from the semaphore's waiting process list and is placed on the ready-active queue. The system scheduler is then called.
- If blocked on a ring buffer (wait-active state), the subject process is removed from the ring buffer's waiting process list and is placed on the ready-active queue. The system scheduler is then called. No adjustment is made for any partial transfer to or from the ring buffer that may have occurred on behalf of the subject process; that is, the buffer is not reset. The system scheduler is then called.
- If the subject process is in the exception-wait-active state, control returns to the caller. The subject process will be placed on the ready-active queue when the exception handler finishes processing the exception (see Chapter 17).

If the subject process has a stop flag in effect and the subject process is not the caller, STOP sets the process's stop-flag byte to TRUE and returns to the caller. If in the ready-suspended, wait-suspended, or exception-wait-suspended state, the subject process is made active and is then treated as described for a process in one of the applicable active states listed above.

This request is implemented through the STPC$ kernel primitive.

Error Returns

See Section 11.2 for general information about error returns. The following exception codes may be returned:

ES$CDN	(type: SYSTEM_SERVICE)—Cannot specify both name and descriptor
ES$IST	(type: SYSTEM_SERVICE)—Illegal structure descriptor; no such process exists

12.8 SUSPEND

MACRO equivalent: SPND$

The SUSPEND function places an active process in the suspended state if no prior RESUME request is pending for that process. The function returns a Boolean TRUE or FALSE value to indicate whether the process was suspended.

This request allows the calling process to suspend either itself or another process. The suspended process is prevented from executing until resumed by another process (see RESUME request).

Syntax

```
SUSPEND ⟦ ( { DESC := process-descriptor }
            { NAME := process-name       }
          ⟦ STATUS := status-record ⟧ ) ⟧
```

process-descriptor
The identifier of a variable of predefined type PROCESS_DESC that contains the process identifier. The variable must have been previously initialized by a process invocation or by an INIT_PROCESS_DESC request.

process-name
A character-string constant or the identifier of a variable of predefined type NAME_STR that contains the 6-character name of an existing process (see Section 11.1.2).

status-record
The identifier of a variable of predefined record type EXC_STATUS th may receive an exception type and code. If you specify this parameter, the exception status, either success or error, that results from issuing the request is reported in this variable. Otherwise, an error causes the corresponding exception to be reported. The format of the exception record is described in Section 11.1.2.

If you specify neither a process-descriptor nor a process-name parameter, the procedure suspends the process issuing the request.

Example

```
TYPE
  Lines = 0..15;

VAR
  Gate : SEMAPHORE_DESC;
  Line2_desc : PROCESS_DESC;

[PRIORITY(10), STACK_SIZE(100)] PROCESS Control (n : Lines);
BEGIN

  (* If SUSPEND is TRUE, suspend this process. When (and if)
     this process is resumed, THEN signal the semaphore Gate.
     If SUSPEND is false, then take ELSE path and continue. *)
  IF SUSPEND
   THEN SIGNAL (DESC := Gate)
   ELSE (* Continue, but note that the suspend count is
           decremented. *);
  (* Unconditionally decrement the suspend count of the process
     whose id is in p2_desc. *)
  IF SUSPEND (DESC := Line2_desc)
   THEN
   ELSE;

  (* Unconditionally decrement the suspend count of the process
     whose name is 'LINE3 '. *)
  IF SUSPEND (NAME := 'LINE3 ')
   THEN
   ELSE;

END; (* Process Control *)
```

Semantics

The SUSPEND procedure decrements the suspension count associated with the specified process. If the count changes from 0 to −1, the state of the process is changed to the ready-suspended, wait-suspended, or exception-wait-suspended state, depending on its state at the time of suspension. If the suspended process was the caller, it is removed from the run state, and the highest-priority process in the ready-active state is placed in the run state. Otherwise, control returns to the caller after the count is decremented.

Depending on the value of the suspension count, a SUSPEND request may not suspend a process. The immediate effect of the request is indicated as a Boolean function return. A TRUE return indicates that the process was changed to a suspended state. A FALSE return indicates that the process was not suspended by the current SUSPEND operation.

Note

A SUSPEND operation on a "stopped" process is ignored. (See the STOP request.)

Transitions between the wait-suspended or exception-wait-suspended states and the ready-suspended state can occur while a process is suspended. (A SIGNAL operation can cause a transition from wait-suspended to ready-suspended state, for example.) A RESUME request is required to return a suspended process to one of the active states.

The maximum value of the suspension counter is 32,767; that is, the counter can record a maximum of 32,767 successive SUSPEND or RESUME requests.

This request is implemented through the SPND$ kernel primitive.

Error Returns

See Section 11.2 for general information about error returns. The following exception codes may be returned:

ES$CDN	(type: SYSTEM_SERVICE)—Cannot specify both name and descriptor
ES$IST	(type: SYSTEM_SERVICE)—Illegal structure descriptor; no such process exists

Chapter 13

Binary and Counting Semaphore and Mutex Management Requests

This chapter describes the requests that operate on binary and counting semaphore structures and mutual exclusion (mutex) structures. Those requests, implemented through the predeclared procedures and functions listed in Table 13–1, are the Pascal language interface to the services provided by the kernel's semaphore management primitives. The requests are used by two or more cooperating processes for mutual exclusion and other forms of synchronization.

Table 13–1 summarizes the operations performed by those requests.

Table 13–1: Operations Performed by Binary and Counting Semaphore and Mutex Management Requests

Request	Operation
COND_SIGNAL	Performs a conditional signal operation, which increments the semaphore variable only if a process is waiting on the semaphore. The function returns a FALSE indication if the signal was not performed.
COND_WAIT	Performs a conditional wait operation, which decrements the semaphore variable only if the semaphore has been signaled or is initialized to TRUE—that is, its value is nonzero. Essentially a "test semaphore and decrement if possible" operation, this request never causes the requesting process to block. The function returns a FALSE indication if the wait was not performed.
CREATE_BINARY_SEMAPHORE	Creates a binary semaphore and sets up a descriptor for efficient reference to the semaphore.

Table 13–1 (Cont.): Operations Performed by Binary and Counting Semaphore and Mutex Management Requests

Request	Operation
CREATE_BINARY_SEMAPHORE_P	Creates a binary semaphore and sets up a descriptor for efficient reference to the semaphore by a procedure.
CREATE_COUNTING_SEMAPHORE	Creates a counting semaphore and sets up a descriptor for efficient reference to the semaphore.
CREATE_COUNTING_SEMAPHORE_P	Creates a counting semaphore and sets up a descriptor for efficient reference to the semaphore by a procedure.
CREATE_MUTEX	Creates a mutex structure.
DESTROY	Deletes a structure from the system and deallocates the memory space used by it.
DESTROY_MUTEX	Deletes a specified mutex structure and deallocates the memory space associated with it.
GET_VALUE	Obtains a structure's value and type code.
INIT_STRUCTURE_DESC	Sets up a descriptor for efficient reference to a semaphore.
LOCK_MUTEX	Locks a resource for exclusive use. The process is blocked if the resource is already locked.
SIGNAL	Signals a binary or a counting semaphore and unblocks the first waiting process.
SIGNAL_ALL	Unblocks any and all processes that may be waiting on the specified semaphore.
UNLOCK_MUTEX	Releases a resource for use by other processes.
WAIT	Tests the specified binary or counting semaphore for a signal (positive value). The calling process is blocked if the semaphore was not signaled.
WAIT_ANY	An enhanced form of WAIT that permits a process to test up to four semaphores for the arrival of a signal. The time interval during which a process may be blocked can be specified.

13.1 Binary and Counting Semaphores

A binary semaphore is a variable that can assume the values of 0 and 1. A signal of a binary semaphore whose value is 0 allows exactly one subsequent wait to proceed without blocking the process issuing the wait. Signaling a binary semaphore whose value is 1 has no effect; that is, exactly one process issuing a subsequent wait operation will proceed without blocking.

A counting semaphore uses a variable that can assume a value greater than 1. As with binary semaphores, a signal of a counting semaphore whose value is 0 allows exactly one subsequent wait operation to proceed without blocking the process. Unlike binary semaphores, however, successive signals without intervening wait operations are not lost. Each signal is counted and will allow one wait to proceed without blocking.

You must include in your program or module the definitions of the routines that implement the requests described in this chapter before using them. See Appendix I for more information.

13.2 Mutex Structures

A mutex (mutual exclusion) structure is an optimization of the binary semaphore that allows dynamic processes within the same program (static process) to share a common resource in an orderly and controlled manner. (If the resource is to be shared between static processes, you must protect it by direct use of a binary or counting semaphore.)

The general semantics of mutex operations are identical to the comparable operations on binary semaphores; that is, UNLOCK_MUTEX corresponds to SIGNAL, and LOCK_MUTEX corresponds to WAIT. An important difference is that when a program locks a mutex to gain access to a shared resource, no call to the WAIT procedure is performed unless other processes are waiting for the resource. This results in a significant improvement in efficiency compared with the WAIT and SIGNAL operations on binary semaphores.

The operations on mutex structures are summarized in the table above. Each procedure takes a mutex variable as its argument. This variable is the linkage between the user's processes and the mutex structure.

Processes access a mutex-protected shared resource by calling the LOCK_MUTEX and UNLOCK_MUTEX procedures in paired sequence as shown in the following:

```
Dynamic Process A0      Dynamic Process A1      Dynamic Process A2
        .                       .                       .
        .                       .                       .
 LOCK_MUTEX (res);       LOCK_MUTEX (res);       LOCK_MUTEX (res);
        .                       .                       .
        .                       .                       .
    use res                 use res                 use res
        .                       .                       .
        .                       .                       .
UNLOCK_MUTEX (res);     UNLOCK_MUTEX (res);     UNLOCK_MUTEX (res);
        .                       .                       .
        .                       .                       .
```

Whenever a process has access to a mutex-protected shared resource, other processes attempting to lock it are blocked waiting until the current user unlocks it. When the resource is unlocked by the process currently using the resource, it is locked by the next process in the resource's waiting-process list.

13.3 COND_SIGNAL

MACRO equivalent: SGLC$

The COND_SIGNAL function, the conditional form of the SIGNAL procedure, signals a specified binary or counting semaphore if at least one process is waiting on that semaphore. The first process waiting on the semaphore is unblocked, and the function returns a Boolean TRUE value. If no process is waiting, the semaphore is not signaled, and the function returns a Boolean FALSE value.

COND_SIGNAL permits the calling process to signal another process that an event it is waiting on has occurred, but only if the request to wait is issued before the signal. This feature allows the caller to selectively signal one of a set of semaphores, unblocking one process, if any, waiting on some semaphore of that set.

Syntax

```
COND_SIGNAL ( { DESC := sem-descriptor }
              { NAME := sem-name       }
             [[ STATUS := status-record ]] )
```

sem-descriptor
The identifier of a variable of predefined type SEMAPHORE_DESC that contains the semaphore structure identifier. The variable must have been previously initialized by a CREATE_BINARY_SEMAPHORE, CREATE_COUNTING_SEMAPHORE, or INIT_STRUCTURE_DESC request.

sem-name
A character-string constant or a variable of predefined type NAME_STR that specifies the 6-character name of an existing semaphore (see Section 11.1.1.1).

status-record
The identifier of a variable of predefined record type EXC_STATUS that may receive an exception type and code. If you specify this parameter, the exception status, either success or error, that results from issuing the request is reported in this variable. Otherwise, an error causes the corresponding exception to be reported. The format of the exception record is described in Section 11.1.2.

Restriction

The semaphore must not be a queue semaphore.

Example

```
VAR
  Gate : SEMAPHORE_DESC;
  Someone_waiting : BOOLEAN;
  N, Count : UNSIGNED;

PROCEDURE Fill_buffer;
BEGIN
   .
   .
   .
END; (* Procedure Fill_buffer *)
```

```
PROCEDURE Compute (Term : UNSIGNED);
BEGIN
  .
  .
  .
END; (* Procedure Compute *)

[PRIORITY(10), STACK_SIZE(100), NAME ('P1    ')] PROCESS P1;
BEGIN

  (* Conditionally signal an unnamed semaphore. *)
  IF COND_SIGNAL (DESC := Gate)
   THEN Count := Count + 1  (* Keep count of the signals. *)
   ELSE Fill_buffer;

  (* Conditionally signal a named semaphore. *)
  Someone_waiting := COND_SIGNAL (NAME := 'ACCESS');
  IF Someone_waiting
   THEN Count := Count + 1  (* Keep count of the signals. *)
   ELSE Compute (N+1);  (* Compute next term. *)

END; (* Process P1 *)
```

Semantics

The COND_SIGNAL function signals the specified semaphore only if at least one process is waiting. Otherwise, it returns a Boolean FALSE value to indicate that the semaphore was not signaled. If the signal operation succeeds, COND_SIGNAL unblocks the first waiting process, decrements the semaphore value, and calls the scheduler, if required. The calling process may be preempted; the process loses control of the CPU. The function will return a Boolean TRUE value, indicating that the semaphore was signaled, when control returns to the caller.

This request is implemented through the SGLC$ kernel primitive.

Error Returns

See Section 11.2 for general information about error returns. The following exception codes may be returned:

ES$CDN	(type: SYSTEM_SERVICE)—Cannot specify both descriptor and name
ES$IST	(type: SYSTEM_SERVICE)—Illegal structure descriptor; no such semaphore exists
ES$MDN	(type: SYSTEM_SERVICE)—Must specify descriptor or name

13.4 COND_WAIT

MACRO equivalent: WAIC$

The COND_WAIT function, the conditional form of the WAIT procedure, permits the calling process to test for the arrival of a signal from another process without being blocked when no signal has occurred. If the semaphore has been signaled, COND_WAIT returns a Boolean TRUE value and closes the semaphore. If the semaphore has not been signaled, the function returns a Boolean FALSE value. In neither case is the calling process blocked, that is, made to wait until the semaphore is signaled. (Compare with the WAIT request.)

This request allows more process concurrency than is possible with the totally synchronous operation provided by the WAIT request. (See also SIGNAL and COND_SIGNAL.)

Syntax

```
COND_WAIT (  { DESC := sem-descriptor  }
             { NAME := sem-name        }
            [[ STATUS := status-record ]] )
```

sem-descriptor
The identifier of a variable of predefined type SEMAPHORE_DESC that contains the semaphore structure identifier. The variable must have been previously initialized by a CREATE_BINARY_SEMAPHORE, CREATE_COUNTING_SEMAPHORE, or INIT_STRUCTURE_DESC request.

sem-name
A character-string constant or a variable of predefined type NAME_STR that specifies the 6-character name of an existing semaphore (see Section 11.1.1.1).

status-record
The identifier of a variable of predefined record type EXC_STATUS that may receive an exception type and code. If you specify this parameter, the exception status, either success or error, that results from issuing the request is reported in this variable. Otherwise, an error causes the corresponding exception to be reported. The format of the exception record is described in Section 11.1.2.

Restriction

The specified semaphore must not be a queue semaphore.

Example

```
TYPE
  Block_number = 0..511;

VAR
  Gate : SEMAPHORE_DESC;
  N : Block_number;
```

```
PROCEDURE Output_block (I : Block_number);
BEGIN
   .
   .
   .
END; (* Procedure Output_block *)

[PRIORITY(10), STACK_SIZE(100), NAME ('P1    ')] PROCESS P1;
BEGIN

  (* Wait on an unnamed semaphore. *)
  IF COND_WAIT (DESC := Gate)
   THEN BEGIN
          Output_block (N);
          N := N + 1;  (* Increment block number. *)
          SIGNAL (DESC := Gate);
        END;

  (* Wait on a named semaphore. *)
  IF COND_WAIT (NAME := 'ACCESS')
   THEN BEGIN
          Output_block (N);
          N := N + 1;  (* Increment block number. *)
          SIGNAL (DESC := Gate);
        END;

END; (* Process P1 *)
```

Semantics

The COND_WAIT function decrements the specified semaphore variable if its value is greater than 0 and returns a Boolean TRUE value. If the semaphore value is 0, COND_WAIT returns a Boolean FALSE value.

This request is implemented through the WAIC$ kernel primitive.

Error Returns

See Section 11.2 for general information about error returns. The following exception codes may be returned:

ES$CDN	(type: SYSTEM_SERVICE)—Cannot specify both descriptor and name
ES$IST	(type: SYSTEM_SERVICE)—Illegal structure descriptor; no such semaphore exists
ES$MDN	(type: SYSTEM_SERVICE)—Must specify descriptor or name

13.5 CREATE_BINARY_SEMAPHORE

MACRO equivalent: CRST$

The CREATE_BINARY_SEMAPHORE function creates a binary semaphore structure. If the semaphore is successfully created, the function returns a Boolean TRUE value. If not enough free system memory is available to create the semaphore, the function returns a Boolean FALSE value.

The function permits a process to create a binary semaphore that can be manipulated by the various semaphore management requests described in this chapter.

Syntax

```
CREATE_BINARY_SEMAPHORE ( ⟦ WAIT_ORDER := { FIFO } ⟧
                                            { PRIO }
                          ⟦ VALUE := gate-count ⟧
                          { DESC := sem-descriptor }
                          { NAME := sem-name       }
                          ⟦ STATUS := status-record ⟧ )
```

WAIT_ORDER
The order in which waiting processes are queued on the semaphore's waiting process list. FIFO (first-in/first-out) is the default value. PRIO specifies ordering by process priority.

gate-count
The identifier of a variable or a constant of predefined type BIN_SEM_VAL that specifies the initial value of the semaphore, either 0 or 1. A value of 0, the default, specifies that the semaphore is closed; a value of 1 specifies that the semaphore is open.

sem-descriptor
The identifier of a variable of predefined type SEMAPHORE_DESC that is to receive the semaphore's structure identifier.

sem-name
A character-string constant or a variable of predefined type NAME_STR that specifies the 6-character name of the semaphore (see Section 11.1.1.1). You must not use the name of an existing process or structure.

status-record
The identifier of a variable of predefined record type EXC_STATUS that may receive an exception type and code. If you specify this parameter, the exception status, either success or error, that results from issuing the request is reported in this variable. Otherwise, an error causes the corresponding exception to be reported. The format of the exception record is described in Section 11.1.2.

If you do not specify the sem-name parameter, the function creates an unnamed binary semaphore identified by the information returned in the sem-descriptor variable.

Example

```
%INCLUDE 'EXC.PAS'

VAR
  Access, Gate_A, Gate_B : SEMAPHORE_DESC;

[INITIALIZE] PROCEDURE Init;
(* Create the needed binary semaphores.  If any create fails, then
   report an exception. *)
BEGIN

  (* Create an unnamed binary semaphore, which is initially closed
     and has FIFO ordering. *)
  IF NOT CREATE_BINARY_SEMAPHORE (DESC := Gate_A)
   THEN REPORT (EXC_TYPE := [RESOURCE], EXC_CODE := ES$NMK);

  (* Create a  named binary semaphore,  which is  initially closed
     and has FIFO ordering. *)
  IF NOT CREATE_BINARY_SEMAPHORE (DESC := Access, NAME := 'ACCESS')
   THEN REPORT (EXC_TYPE := [RESOURCE], EXC_CODE := ES$NMK);

  (* Create an  unnamed binary semaphore, which is  initially open
     and has FIFO ordering. *)
  IF NOT CREATE_BINARY_SEMAPHORE (DESC := Gate_B, VALUE := 1)
   THEN REPORT (EXC_TYPE := [RESOURCE], EXC_CODE := ES$NMK);

END; (* Procedure Init *)
```

Semantics

The CREATE_BINARY_SEMAPHORE function requests the kernel to allocate and initialize a binary semaphore structure in system-common memory.

If the semaphore is successfully created, the function returns a Boolean TRUE value. The semaphore is named as specified in the sem-name parameter, and its structure identifier is copied into the variable specified by the sem-descriptor parameter.

If the semaphore cannot be created because the system's free-memory pool has insufficient space, the function returns a Boolean FALSE value.

This request is implemented through the CRST$ kernel primitive.

Error Returns

See Section 11.2 for general information about error returns. The following exception codes may be returned:

ES$MDN	(type: SYSTEM_SERVICE)—Must specify structure descriptor or name
ES$SNI	(type: SYSTEM_SERVICE)—Structure name already in use; the specified name has already been given to another process, semaphore, ring buffer, logical name, or shared region

The request may return the following errors, though not as a result of standard Pascal programming practice:

ES$IAD	(type: SYSTEM_SERVICE)—Invalid address; pointer to descriptor is odd or not in user address space
ES$IPM	(type: SYSTEM_SERVICE)—Illegal parameter; an invalid structure type or argument value was specified
ES$IPR	(type: SYSTEM_SERVICE)—Illegal primitive; user code attempted to create a PCB (type ST.PCB specified)

13.6 CREATE_BINARY_SEMAPHORE_P

MACRO equivalent: none

CREATE_BINARY_SEMAPHORE_P creates a binary semaphore structure by a procedure. If the semaphore is successfully created, the STATUS parameter is set to ES$NOR. If not enough free system memory is available to create the semaphore, the STATUS parameter is set to the appropriate exception code.

The procedure permits a process to create a binary semaphore that can be manipulated by the semaphore management requests described in this chapter.

Syntax

```
                                  ⟦                    { FIFO }  ⟧
CREATE_BINARY_SEMAPHORE_P (       ⟦  WAIT_ORDER :=     { PRIO }  ⟧
                                  ⟦ VALUE := gate-count ⟧
                                  { DESC := sem-descriptor }
                                  { NAME := sem-name       }
                                  ⟦ STATUS := status-record ⟧ )
```

WAIT_ORDER

The order in which waiting processes are queued on the semaphore's waiting process list. FIFO (first-in/first-out) is the default value. PRIO specifies ordering by process priority.

gate-count

The identifier of a variable or a constant of predefined type BIN_SEM_VAL that specifies the initial value of the semaphore, either 0 or 1. A value of 0, the default, specifies that the semaphore is closed; a value of 1 specifies that the semaphore is open.

sem-descriptor

The identifier of a variable of predefined type SEMAPHORE_DESC that is to receive the semaphore's structure identifier.

sem-name

A character-string constant or a variable of predefined type NAME_STR that specifies the 6-character name of the semaphore (see Section 11.1.1.1). You must not use the name of an existing process or structure.

status-record

The identifier of a variable of predefined record type EXC_STATUS that may receive an exception type and code. If you specify this parameter, the exception status, either success or error, that results from issuing the request is reported in this variable. Otherwise, an error causes the corresponding exception to be reported. The format of the exception record is described in Section 11.1.2.

If you do not specify the sem-name parameter, the function creates an unnamed binary semaphore identified by the information returned in the sem-descriptor variable.

Example

```
%INCLUDE 'EXC.PAS'
%INCLUDE 'CRPROC.PAS'

VAR
   BDESC1, BDESC2 : SEMAPHORE_DESC;
   P_STATUS : EXC_STATUS;
   SUCCESS : Boolean;

(* Create the binary semaphores.  If any create fails, then
   set Boolean SUCCESS to false. *)
[INITIALIZE] PROCEDURE Init;
BEGIN
   SUCCESS := True;
   CREATE_BINARY_SEMAPHORE_P (WAIT_ORDER := FIFO, VALUE := 0,
                              DESC := BDESC1, NAME := 'IMFREE',
                              STATUS := P_STATUS);
   IF (P_STATUS.EXC_CODE <> ES$NOR)
                              THEN SUCCESS := False;
   CREATE_BINARY_SEMAPHORE_P (DESC := BDESC2, STATUS := P_STATUS);
   IF (P_STATUS.EXC_CODE <> ES$NOR)
                              THEN SUCCESS := False;
END;

BEGIN (* Main *)
   IF NOT SUCCESS
      THEN WRITELN('%ERROR - Semaphore creation failed')
      ELSE
      .
      .
      .
END.
```

Semantics

The CREATE_BINARY_SEMAPHORE_P procedure requests the kernel to allocate and initialize a binary semaphore structure in system-common memory.

If the semaphore is successfully created, the STATUS parameter is set to ES$NOR. The semaphore is named as specified in the sem-name parameter, and its structure identifier is copied into the variable specified by the sem-descriptor parameter.

If the semaphore cannot be created because the system's free-memory pool has insufficient space, the STATUS parameter is set to the appropriate exception code.

This request is implemented through the CRST$ kernel primitive.

Error Returns

See Section 11.2 for general information about error returns. The following exception codes may be returned:

ES$MDN	(type: SYSTEM_SERVICE)—Must specify structure descriptor or name
ES$SNI	(type: SYSTEM_SERVICE)—Structure name already in use; the specified name has already been given to another process, semaphore, ring buffer, logical name, or shared region

The request may return the following errors, though not as a result of standard Pascal programming practice:

ES$IAD	(type: SYSTEM_SERVICE)—Invalid address; pointer to descriptor is odd or not in user address space
ES$IPM	(type: SYSTEM_SERVICE)—Illegal parameter; an invalid structure type or argument value was specified
ES$IPR	(type: SYSTEM_SERVICE)—Illegal primitive; user code attempted to create a PCB (type ST.PCB specified)

13.7 CREATE_COUNTING_SEMAPHORE

MACRO equivalent: CRST$

The CREATE_COUNTING_SEMAPHORE function creates a counting semaphore structure. If the semaphore is successfully created, the function returns a Boolean TRUE value. If insufficient memory exists to create the semaphore or if an exception occurs, the function returns a Boolean FALSE value.

The function permits a process to create a counting semaphore that can be manipulated by the various semaphore management requests described in this chapter.

Syntax

```
CREATE_COUNTING_SEMAPHORE ( [[ WAIT_ORDER := { FIFO } ]]
                                             { PRIO }
                            [[ VALUE := gate-count ]]
                            { DESC := sem-descriptor }
                            { NAME := sem-name       }
                            [[ STATUS := status-record ]] )
```

WAIT_ORDER

The order in which waiting processes are queued on the semaphore's waiting process list. FIFO (first-in/first-out) is the default value; PRIO specifies ordering by process priority.

gate-count

A constant or variable that specifies the initial value of the semaphore. This value must be from 0 to 65,535. A value of 0, the default, specifies that the semaphore is closed. A nonzero value specifies that the semaphore is open and indicates the number of WAIT or COND_WAIT requests that processes can perform before the semaphore closes.

sem-descriptor

The identifier of a variable of predefined type SEMAPHORE_DESC that is to receive the semaphore's structure identifier.

sem-name

A character-string constant or a variable of predefined type NAME_STR that specifies the 6-character name of the semaphore (see Section 11.1.1.1). You must not use the name of an existing process or structure.

status-record

The identifier of a variable of predefined record type EXC_STATUS that may receive an exception type and code. If you specify this parameter, the exception status, either success or error, that results from issuing the request is reported in this variable. Otherwise, an error causes the corresponding exception to be reported. The format of the exception record is described in Section 11.1.2.

If you do not specify a sem-name, the function creates an unnamed counting semaphore identified by the information returned in the sem-descriptor variable.

Example

```
%INCLUDE 'EXC.PAS'

VAR
  Access, Gate, Priority_access : SEMAPHORE_DESC;

[INITIALIZE] PROCEDURE Init;
(* Create the needed  counting semaphores.  If any  create fails,
   then report an exception. *)
BEGIN

  (* Create a named counting semaphore, which is initially closed
     and has FIFO ordering. *)
  IF NOT CREATE_COUNTING_SEMAPHORE (DESC := Access, NAME := 'ACCESS')
   THEN REPORT (EXC_TYPE := [RESOURCE], EXC_CODE := ES$NMK);

  (* Create  an unnamed  counting  semaphore,  which is initially
     open and has FIFO ordering. *)
  IF NOT CREATE_COUNTING_SEMAPHORE (DESC := Gate, VALUE := 1)
   THEN REPORT (EXC_TYPE := [RESOURCE], EXC_CODE := ES$NMK);

  (* Create an  unnamed counting  semaphore,  which is  initially
     closed and has priority ordering. *)
  IF NOT CREATE_COUNTING_SEMAPHORE (DESC := Priority_access, WAIT_ORDER := PRIO)
   THEN REPORT (EXC_TYPE := [RESOURCE], EXC_CODE := ES$NMK);

END; (* Procedure Init *)
```

Semantics

The CREATE_COUNTING_SEMAPHORE function requests the kernel to allocate and initialize a counting semaphore structure in system-common memory.

If the semaphore is successfully created, the function returns a Boolean TRUE value. The semaphore is named as specified in the sem-name parameter, and its structure identifier is copied into the variable specified by the sem-descriptor parameter.

If the semaphore cannot be created because the system's free-memory pool has insufficient space, the function returns a Boolean FALSE value.

This request is implemented through the CRST$ kernel primitive.

Error Returns

See Section 11.2 for general information about error returns. The following exception codes may be returned:

ES$MDN	(type: SYSTEM_SERVICE)—Must specify structure descriptor or name
ES$SNI	(type: SYSTEM_SERVICE)—Structure name already in use; the specified name has already been given to another process, semaphore, ring buffer, logical name, or shared region

The request may return the following errors, though not as a result of standard Pascal programming practice:

ES$IAD	(type: SYSTEM_SERVICE)—Invalid address; pointer to descriptor is odd or not in user address space
ES$IPM	(type: SYSTEM_SERVICE)—Illegal parameter; an invalid structure type or argument value was specified
ES$IPR	(type: SYSTEM_SERVICE)—Illegal primitive; user code attempted to create a PCB (type ST.PCB specified)

13.8 CREATE_COUNTING_SEMAPHORE_P

MACRO equivalent: none

CREATE_COUNTING_SEMAPHORE_P creates a counting semaphore structure by a procedure. If the semaphore is successfully created, the STATUS parameter is set to ES$NOR. If insufficient memory exists to create the semaphore or if an exception occurs, the STATUS parameter is set to the appropriate exception code.

The procedure permits a process to create a counting semaphore that can be manipulated by the various semaphore management requests described in this chapter.

Syntax

```
CREATE_COUNTING_SEMAPHORE_P ( ⟦ WAIT_ORDER := { FIFO } ⟧
                                              { PRIO }
                              ⟦ VALUE := gate-count ⟧
                              { DESC := sem-descriptor }
                              { NAME := sem-name       }
                              ⟦ STATUS := status-record ⟧ )
```

WAIT_ORDER
The order in which waiting processes are queued on the semaphore's waiting process list. FIFO (first-in/first-out) is the default value; PRIO specifies ordering by process priority.

gate-count
A constant or variable that specifies the initial value of the semaphore. This value must be from 0 to 65,535. A value of 0, the default, specifies that the semaphore is closed. A nonzero value specifies that the semaphore is open and indicates the number of WAIT or COND_WAIT requests that processes can perform before the semaphore closes.

sem-descriptor
The identifier of a variable of predefined type SEMAPHORE_DESC that is to receive the semaphore's structure identifier.

sem-name
A character-string constant or a variable of predefined type NAME_STR that specifies the 6-character name of the semaphore (see Section 11.1.1.1). You must not use the name of an existing process or structure.

status-record
The identifier of a variable of predefined record type EXC_STATUS that may receive an exception type and code. If you specify this parameter, the exception status, either success or error, that results from issuing the request is reported in this variable. Otherwise, an error causes the corresponding exception to be reported. The format of the exception record is described in Section 11.1.2.

If you do not specify a sem-name, the function creates an unnamed counting semaphore identified by the information returned in the sem-descriptor variable.

Example

```
%INCLUDE 'EXC.PAS'
%INCLUDE 'CRPROC.PAS'

VAR
   P_STATUS : EXC_STATUS;
   SUCCESS := Boolean;
   SYNC1, SYNC2 : SEMAPHORE_DESC;

(* Create the counting semaphores.  If any  create fails,
   then set Boolean SUCCESS to false. *)
[INITIALIZE] PROCEDURE Init;
BEGIN
   SUCCESS := True
   CREATE_COUNTING_SEMAPHORE_P (WAIT_ORDER := FIFO, VALUE := 0,
                                DESC := SYNC1, NAME := 'GATE   ',
                                STATUS := P_STATUS);
   IF (P_STATUS.EXC_CODE <> ES$NOR)
                                THEN SUCCESS := False;
   CREATE_COUNTING_SEMAPHORE_P (NAME = 'CS1    ', STATUS := P_STATUS);
   IF (P_STATUS.EXC_CODE <> ES$NOR)
                                THEN SUCCESS := False;
END;

BEGIN (* Main *)
   IF NOT SUCCESS
      THEN WRITELN('%ERROR - Semaphore creation failed')
      ELSE
      .
      .
      .
END.
```

Semantics

The CREATE_COUNTING_SEMAPHORE_P procedure requests the kernel to allocate and initialize a counting semaphore structure in system-common memory.

If the semaphore is successfully created, the STATUS parameter is set to ES$NOR. The semaphore is named as specified in the sem-name parameter, and its structure identifier is copied into the variable specified by the sem-descriptor parameter.

If the semaphore cannot be created because the system's free-memory pool has insufficient space, the STATUS parameter is set to the appropriate exception code.

This request is implemented through the CRST$ kernel primitive.

Error Returns

See Section 11.2 for general information about error returns. The following exception codes may be returned:

ES$MDN	(type: SYSTEM_SERVICE)—Must specify structure descriptor or name
ES$SNI	(type: SYSTEM_SERVICE)—Structure name already in use; the specified name has already been given to another process, semaphore, ring buffer, logical name, or shared region

The request may return the following errors, though not as a result of standard Pascal programming practice:

ES$IAD	(type: SYSTEM_SERVICE)—Invalid address; pointer to descriptor is odd or not in user address space
ES$IPM	(type: SYSTEM_SERVICE)—Illegal parameter; an invalid structure type or argument value was specified
ES$IPR	(type: SYSTEM_SERVICE)—Illegal primitive; user code attempted to create a PCB (type ST.PCB specified)

13.9 CREATE_MUTEX

MACRO equivalent: CRST$

The CREATE_MUTEX procedure creates a mutex (mutual exclusion) structure for use in guarding the access to a shared resource. The mutex is initialized to the UNLOCK state.

Syntax

```
CREATE_MUTEX ( MUTEX_VAR := mutex-variable-id
               [[ WAIT_ORDER := { FIFO } ]]
                                { PRIO }
               [[ STATUS := status-record ]] )
```

mutex-variable-id

The identifier of a variable of predefined type mutex that will be initialized with the mutex identification and control information.

WAIT_ORDER

The ordering of the list of processes waiting to use the mutex-protected resource. FIFO specifies first-in-first-out order and is the default value. PRIO specifies ordering by process priority.

status-record

The identifier of a variable of predefined record type EXC_STATUS that may receive an exception type and code. If you specify this parameter, the exception status, either success or error, that results from issuing the request is reported in this variable. Otherwise, an error causes the corresponding exception to be reported. The format of the exception record is described in Section 11.1.2.

Restriction

The processes using the mutex must reside in the same program (static process).

Example

```
%INCLUDE 'MUTEX.PAS'

VAR
  M1, M2 : MUTEX;

[INITIALIZE] PROCEDURE Init;
BEGIN

  (* Create a MUTEX with FIFO wait order. *)
  CREATE_MUTEX (MUTEX_VAR := M1);

  (* Create a MUTEX with PRIO wait order. *)
  CREATE_MUTEX (MUTEX_VAR := M2, WAIT_ORDER := PRIO);

END; (* Procedure Init *)
```

Semantics

The CREATE_MUTEX procedure creates a binary semaphore, initializes the variable specified by the mutex-variable-id parameter, and returns to the caller. The binary semaphore is initialized to the UNLOCK state.

This request is implemented through the CRST$ kernel primitive.

Error Returns

See Section 11.2 for general information about error returns. The following exception code may be returned:

ES$NMK	(type: RESOURCE)—Insufficient space for kernel structure; could not obtain space for the mutex structure

The request may return the following errors, though not as a result of standard Pascal programming practice:

ES$IPM	(type: SYSTEM_SERVICE)—Illegal parameter
ES$IPR	(type: SYSTEM_SERVICE)—Illegal primitive
ES$SNI	(type: SYSTEM_SERVICE)—Structure name already in use

13.10 DESTROY

MACRO equivalent: DLST$

The DESTROY procedure deletes a specified structure (in this case, a binary or counting semaphore) from the system and deallocates the memory space associated with the structure. The operation is performed only if no processes are blocked on the structure at the time of the call.

Syntax

$$\text{DESTROY (}\ \left\{\begin{array}{l}\text{DESC := descriptor}\\ \text{NAME := name}\end{array}\right\}$$
$$[\![\ \text{STATUS := status-record}\ ]\!]\ )$$

descriptor

The identifier of a variable of predefined type SEMAPHORE_DESC that contains the semaphore's structure identifier. The variable must have been previously initialized by a CREATE_BINARY_SEMAPHORE, CREATE_COUNTING_SEMAPHORE, or INIT_STRUCTURE_DESC request.

name

A character-string constant or a variable of predefined type NAME_STR that specifies the 6-character name of an existing semaphore.

status-record

The identifier of a variable of predefined record type EXC_STATUS that may receive an exception type and code. If you specify this parameter, the exception status, either success or error, that results from issuing the request is reported in this variable. Otherwise, an error causes the corresponding exception to be reported. The format of the exception record is described in Section 11.1.2.

Example

```
VAR
  Gate : SEMAPHORE_DESC;

[TERMINATE] PROCEDURE Term;
BEGIN

  (* Destroy an unnamed semaphore. *)
  DESTROY (DESC := Gate);

  (* Destroy a named semaphore. *)
  DESTROY (NAME := 'GATE  ');

END; (* Procedure Term *)
```

Semantics

If a semaphore is not in use, DESTROY removes the semaphore's name (if one exists) from the system name table, returns the space that the semaphore occupies to the system's free-memory pool, and returns control to the caller.

This request is implemented through the DLST$ kernel primitive.

Error Returns

See Section 11.2 for general information about error returns. The following exception codes may be returned:

ES$CDN	(type: SYSTEM_SERVICE)—Cannot specify both descriptor and name
ES$IST	(type: SYSTEM_SERVICE)—Illegal structure descriptor; no such semaphore exists
ES$MDN	(type: SYSTEM_SERVICE)—Must specify descriptor or name
ES$SIU	(type: SYSTEM_SERVICE)—Structure is in use

13.11 DESTROY_MUTEX

MACRO equivalent: DLST$

The DESTROY_MUTEX procedure deletes a mutex structure from the system and deallocates the memory space associated with it. The operation is performed only if there are no processes blocked on the mutex at the time of the call.

Syntax

```
DESTROY_MUTEX (  MUTEX_VAR := mutex-variable-id
                 [[ STATUS := status-record ]] )
```

mutex-variable-id

The identifier of a variable of predefined type mutex that is the mutex variable of the mutex to be destroyed. The variable must have been previously initialized by the CREATE_MUTEX procedure.

status-record

The identifier of a variable of predefined record type EXC_STATUS that may receive an exception type and code. If you specify this parameter, the exception status, either success or error, that results from issuing the request is reported in this variable. Otherwise, an error causes the corresponding exception to be reported. The format of the exception record is described in Section 11.1.2.

Restriction

The processes using the mutex must reside in the same program (static process).

Example

```
%INCLUDE 'MUTEX.PAS'

VAR
  M1 : MUTEX;

[TERMINATE] PROCEDURE Term;
BEGIN

  (* Destroy the MUTEX. *)
  DESTROY_MUTEX (MUTEX_VAR := M1);

END; (* Procedure Term *)
```

Semantics

If a mutex is not currently in use, DESTROY_MUTEX returns the space that the associated semaphore occupies to the system's free-memory pool, updates the mutex data structure, and returns control to the caller.

When a mutex is deleted with DESTROY_MUTEX, the mutex-variable is set to 0, indicating the mutex is locked. The next operation (LOCK_MUTEX or UNLOCK_MUTEX) will do a wait operation and fail, which helps to prevent operations on mutexes after they have been destroyed.

This request is implemented through the DLST$ kernel primitive.

Error Returns

See Section 11.2 for general information about error returns. The following exception codes may be returned:

ES$IST	(type: SYSTEM_SERVICE)—Illegal structure descriptor; no such semaphore exists
ES$SIU	(type: SYSTEM_SERVICE)—Structure is in use; the mutex is being used by another process and cannot be destroyed

13.12 GET_VALUE

MACRO equivalent: GVAL$

The GET_VALUE procedure obtains the value and type code of a specified structure. The code identifies a structure as a semaphore (binary, counting, or queue) or as a ring buffer. The meaning of the structure's value depends on the structure type. For example, the value of a counting semaphore is the current signal count, whereas the value of a ring buffer is the current element count.

Note

The value of a structure may change immediately after it is inspected. Therefore, the information this request provides must be used cautiously, to prevent the introduction of race conditions.

Syntax

```
GET_VALUE (  VALUE := count
             TYP := structure-type
             { DESC := descriptor }
             { NAME := name       }
             [[ STATUS := status-record ]] )
```

count

The identifier of a variable of type INTEGER that receives the structure's value.

structure-type

The identifier of a variable of type INTEGER that receives the structure's type code.

descriptor

The identifier of a variable of predefined type SEMAPHORE_DESC that contains the semaphore's structure identifier. The variable must have been previously initialized by an appropriate CREATE-type request or an INIT_STRUCTURE_DESC request.

name

A character-string constant or a variable of predefined type NAME_STR that specifies the 6-character name of an existing semaphore.

status-record

The identifier of a variable of predefined record type EXC_STATUS that may receive an exception type and code. If you specify this parameter, the exception status, either success or error, that results from issuing the request is reported in this variable. Otherwise, an error causes the corresponding exception to be reported. The format of the exception record is described in Section 11.1.2.

Structure Type Identification Codes

The type codes and meaning of the values that the procedure can return are:

Structure	Type Code	Meaning of Value Parameter
Binary semaphore	0	The value of the gate variable (0 or 1)
Counting semaphore	1	The count of pending signals (0 or positive)
Queue semaphore	2	The count of pending signals (0 or positive)
Ring buffer	3	The count of data elements in the ring buffer
PCB	4	No meaning
SRD	5	No meaning
Unformatted structure	7	No meaning

Example

```
VAR
  Sem_val, Sem_typ : INTEGER;
  Gate : SEMAPHORE_DESC;

[PRIORITY(10), STACK_SIZE(100)] PROCESS P1;
BEGIN

  (* Get the value of an unnamed semaphore. *)
  GET_VALUE (VALUE := Sem_val, TYP := Sem_typ, DESC := Gate);

  (* Get the value of a named semaphore. *)
  GET_VALUE (VALUE := Sem_val, TYP := Sem_typ, NAME := 'GATE  ');

END; (* Process P1 *)
```

Semantics

The GET_VALUE procedure obtains the type code and value of the specified structure, stores this information in the variables specified in the call, and returns control to the caller.

This request is implemented through the GVAL$ kernel primitive.

Error Returns

See Section 11.2 for general information about error returns. The following exception codes may be returned:

ES$CDN	(type: SYSTEM_SERVICE)—Cannot specify both descriptor and name
ES$IPR	(type: SYSTEM_SERVICE)—Illegal primitive; the descriptor or name parameter is a logical name that does not translate to the name of a structure
ES$IST	(type: SYSTEM_SERVICE)—Illegal structure descriptor; no such semaphore or ring buffer exists
ES$MDN	(type: SYSTEM_SERVICE)—Must specify descriptor or name

13.13 INIT_STRUCTURE_DESC

MACRO equivalent: GVAL$

The INIT_STRUCTURE_DESC procedure copies identifying information about a specified binary or counting semaphore structure into a structure descriptor record. That record provides the kernel with a rapid-access path to a structure referred to in the other semaphore management requests described in this chapter.

When you create a semaphore, you may also use the CREATE_BINARY_SEMAPHORE or the CREATE_COUNTING_SEMAPHORE requests to set up a descriptor.

Syntax

```
INIT_STRUCTURE_DESC ( DESC := descriptor
                      NAME := name
                      [[ STATUS := status-record ]] )
```

descriptor

The identifier of a variable of predefined type SEMAPHORE_DESC that is to receive the semaphore's structure identifier.

name

A character-string constant or a variable of predefined type NAME_STR that specifies the 6-character name of an existing semaphore.

status-record

The identifier of a variable of predefined record type EXC_STATUS that may receive an exception type and code. If you specify this parameter, the exception status, either success or error, that results from issuing the request is reported in this variable. Otherwise, an error causes the corresponding exception to be reported. The format of the exception record is described in Section 11.1.2.

Example

```
VAR
  Gate : SEMAPHORE_DESC;

[PRIORITY(10), STACK_SIZE(100), NAME ('P1    ')] PROCESS P1;
BEGIN

  (* Get the id of the semaphore named 'GATE  '. *)
  INIT_STRUCTURE_DESC (DESC := Gate, NAME := 'GATE  ');

END; (* Process P1 *)
```

Semantics

The INIT_STRUCTURE_DESC procedure requests the kernel to copy the index and serial number information associated with the named structure into the specified descriptor variable in the caller's storage area.

This request is implemented through the GVAL$ kernel primitive.

Error Returns

See Section 11.2 for general information about error returns. The following exception code may be returned:

ES$IST	(type: SYSTEM_SERVICE)—Illegal structure descriptor; semaphore does not exist

13.14 LOCK_MUTEX

MACRO equivalent: WAIT$

The LOCK_MUTEX procedure locks the specified mutex structure.

The UNLOCK_MUTEX request is the logical complement of LOCK_MUTEX. Together, the LOCK_MUTEX and UNLOCK_MUTEX requests provide a means for two or more cooperating processes to implement a variety of mutual-exclusion mechanisms for shared resource protection.

Syntax

LOCK_MUTEX (MUTEX_VAR := mutex-variable-id)

mutex-variable-id

The identifier of a variable of predefined type mutex that specifies the mutex to be locked. The variable must have been previously initialized by the CREATE_MUTEX procedure.

Restrictions

- The processes using the mutex must reside in the same program (static process).
- LOCK_MUTEX must be paired with UNLOCK_MUTEX in the program sequence.

Example

```
%INCLUDE 'MUTEX.PAS'

VAR
  M1 : MUTEX;

PROCESS A;
BEGIN

  (* Lock the resource. *)
  LOCK_MUTEX (MUTEX_VAR := M1);
  .
  .
  .
  (* Unlock the resource. *)
  UNLOCK_MUTEX (MUTEX_VAR := M1);

END; (* Process A *)
```

Semantics

The LOCK_MUTEX procedure locks the specified mutex. If the mutex is not currently locked, the calling process continues to run. If another process has already locked the mutex, the calling process will be blocked on a queue of processes waiting to access the mutex.

This request is implemented through the WAIT$ kernel primitive.

Error Returns

See Section 11.2 for general information about error returns. The request may return the following error, though not as a result of standard Pascal programming practice:

ES$IST	(type: SYSTEM_SERVICE)—Illegal structure descriptor; no such binary semaphore exists

13.15 SIGNAL

MACRO equivalent: SGNL$

The SIGNAL procedure signals a specified binary or counting semaphore, unblocking the first process, if any, waiting on that semaphore. This procedure permits the calling process to signal another process that an event has occurred, whether or not the other process is waiting for the signal. (Compare with COND_SIGNAL, the conditional signal request.)

Together the SIGNAL and WAIT requests provide a means for two or more cooperating processes to implement a variety of synchronization and mutual-exclusion mechanisms. (See also the COND_SIGNAL, WAIT, COND_WAIT, and SIGNAL_ALL requests.)

Syntax

$$\text{SIGNAL (}\ \left\{\begin{array}{l}\text{DESC := sem-descriptor}\\ \text{NAME := sem-name}\end{array}\right\}$$
$$[\![\ \text{STATUS := status-record}\ ]\!]\ \text{)}$$

sem-descriptor

The identifier of a variable of predefined type SEMAPHORE_DESC that contains the semaphore structure identifier. The variable must have been previously initialized by a CREATE_BINARY_SEMAPHORE, CREATE_COUNTING_SEMAPHORE, or INIT_STRUCTURE_DESC request.

sem-name

A character-string constant or a variable of predefined type NAME_STR that specifies the 6-character name of an existing semaphore (see Section 11.1.1.1).

status-record

The identifier of a variable of predefined record type EXC_STATUS that may receive an exception type and code. If you specify this parameter, the exception status, either success or error, that results from issuing the request is reported in this variable. Otherwise, an error causes the corresponding exception to be reported. The format of the exception record is described in Section 11.1.2.

Restriction

The semaphore must not be a queue semaphore.

Example

```
VAR
  Gate : SEMAPHORE_DESC;

[PRIORITY(10), STACK_SIZE(100), NAME ('PRODUC')] PROCESS Producer;
BEGIN

  (* Signal an unnamed semaphore. *)
  SIGNAL (DESC := Gate);

  (* Signal a named semaphore. *)
  SIGNAL (NAME := 'GATE  ');

END; (* Process Producer *)
```

Semantics

The SIGNAL procedure increments a binary semaphore's gate variable if its current value is 0. If its value is already 1, the procedure returns control to the caller, with no other action.

The SIGNAL procedure increments a counting semaphore's counter; if its previous value was greater than 0, the procedure returns to the caller.

In either case, if the signal causes the semaphore value to change from 0 to 1 and if at least one process is waiting on the semaphore, the procedure unblocks the first waiting process, decrements the semaphore value, and calls the scheduler. That sequence may cause the calling process to be preempted (lose control of the CPU) depending on the relative priority of the process at the head of the semaphore's queue of blocked processes.

If the semaphore value changes from 0 to 1 and if no process is waiting, control returns to the caller.

This request is implemented through the SGNL$ kernel primitive.

Error Returns

See Section 11.2 for general information about error returns. The following exception codes may be returned:

ES$CDN	(type: SYSTEM_SERVICE)—Cannot specify both descriptor and name
ES$IST	(type: SYSTEM_SERVICE)—Illegal structure descriptor; no such semaphore exists
ES$MDN	(type: SYSTEM_SERVICE)—Must specify descriptor or name

13.16 SIGNAL_ALL

MACRO equivalent: SALL$

The SIGNAL_ALL procedure unblocks all processes waiting on a specified binary or a counting semaphore and sets the value of that semaphore to 0. If no process is waiting on the semaphore, the semaphore value is unchanged. SIGNAL_ALL permits the calling process to signal simultaneously all processes that are waiting for the same event to occur.

Together, the WAIT and SIGNAL_ALL requests provide a means for two or more processes to synchronize on a single event signaled by another process.

Syntax

```
SIGNAL_ALL ( { DESC := sem-descriptor }
             { NAME := sem-name       }
           [[ STATUS := status-record ]] )
```

sem-descriptor

The identifier of a variable of predefined type SEMAPHORE_DESC that contains the semaphore structure identifier. The variable must have been previously initialized by a CREATE_BINARY_SEMAPHORE, CREATE_COUNTING_SEMAPHORE, or INIT_STRUCTURE_DESC request.

sem-name

A character-string constant or a variable of predefined type NAME_STR that specifies the 6-character name of an existing semaphore (see Section 11.1.1.1).

status-record

The identifier of a variable of predefined record type EXC_STATUS that may receive an exception type and code. If you specify this parameter, the exception status, either success or error, that results from issuing the request is reported in this variable. Otherwise, an error causes the corresponding exception to be reported. The format of the exception record is described in Section 11.1.2.

Restriction

The specified semaphore must not be a queue semaphore.

Example

```
VAR
  Access : SEMAPHORE_DESC;

[PRIORITY(10), STACK_SIZE(100), NAME ('DRIVER')] PROCESS Driver;
BEGIN

  (* Signal all processes waiting on an unnamed semaphore. *)
  SIGNAL_ALL (DESC := Access);

  (* Signal all processes waiting on a named semaphore. *)
  SIGNAL_ALL (NAME := 'ACCESS');

END; (* Process Driver *)
```

Semantics

The SIGNAL_ALL procedure unblocks all processes waiting on the specified semaphore and calls the scheduler if any processes are unblocked; the value of the semaphore is set to zero.

If no process is waiting on the specified semaphore, the procedure returns control to the caller, with the semaphore variable unchanged.

The SIGNAL_ALL procedure may cause the calling process to be preempted (lose control of the CPU).

This request is implemented through the SALL$ kernel primitive.

Error Returns

See Section 11.2 for general information about error returns. The following exception codes may be returned:

ES$CDN	(type: SYSTEM_SERVICE)—Cannot specify both descriptor and name
ES$IST	(type: SYSTEM_SERVICE)—Illegal structure descriptor; no such semaphore exists
ES$MDN	(type: SYSTEM_SERVICE)—Must specify descriptor or name

13.17 UNLOCK_MUTEX

MACRO equivalent: SGNL$

The UNLOCK_MUTEX procedure unlocks the specified mutex structure.

The LOCK_MUTEX request is the logical complement of UNLOCK_MUTEX. Together, the LOCK_MUTEX and UNLOCK_MUTEX requests provide a means for two or more cooperating processes to implement a variety of mutual-exclusion mechanisms for shared resource protection.

Syntax

UNLOCK_MUTEX (MUTEX_VAR := mutex-variable-id)

mutex-variable-id

The identifier of a variable of predefined type mutex that specifies the mutex to be locked. The variable must have been previously initialized by the CREATE_MUTEX procedure.

Restrictions

- The processes using the mutex must reside in the same program (static process).
- UNLOCK_MUTEX must be paired with LOCK_MUTEX in the program sequence.

Example

```
%INCLUDE 'MUTEX.PAS'

VAR
  M1 : MUTEX;

PROCESS A;
BEGIN

  (* Lock the resource. *)
  LOCK_MUTEX (MUTEX_VAR := M1);
  .
  .
  .
  (* Unlock the resource. *)
  UNLOCK_MUTEX (MUTEX_VAR := M1);

END; (* Process A *)
```

Semantics

The UNLOCK_MUTEX procedure unlocks the specified mutex. If at least one process is waiting on the mutex, the procedure unblocks the first waiting process and calls the scheduler. This procedure may cause the calling process to be preempted (lose control of the CPU) depending on the relative priority of the process at the head of the mutex's queue of blocked processes.

This request is implemented through the SGNL$ kernel primitive.

Error Returns

See Section 11.2 for general information about error returns. The request may return the following error, though not as a result of standard Pascal programming practice:

ES$IST	(type: SYSTEM_SERVICE)—Illegal structure descriptor; no such binary semaphore exists

13.18 WAIT

MACRO equivalent: WAIT$

The WAIT procedure tests a semaphore for the arrival of a signal from another process. If no signal has occurred (semaphore value is 0), the process is blocked on that semaphore. If a signal has occurred (nonzero semaphore value), the calling process proceeds.

WAIT permits the calling process to receive a signal from another process, indicating that an event on which the calling process is dependent has occurred. For example, a related process that depends on exclusive access to a shared structure can wait on a semaphore that indicates the structure's availability.

Together, the WAIT, SIGNAL, and SIGNAL_ALL requests and their conditional forms provide a means for two or more cooperating processes to implement a variety of synchronization and mutual-exclusion mechanisms.

Syntax

$$\text{WAIT (} \left\{ \begin{array}{l} \text{DESC := sem-descriptor} \\ \text{NAME := sem-name} \end{array} \right\}$$
$$[\![\text{ STATUS := status-record }]\!] \text{)}$$

sem-descriptor

The identifier of a variable of predefined type SEMAPHORE_DESC that contains the semaphore structure identifier. The variable must have been previously initialized by a CREATE_BINARY_SEMAPHORE, CREATE_COUNTING_SEMAPHORE, or INIT_STRUCTURE_DESC request.

sem-name

A character-string constant or a variable of predefined type NAME_STR that specifies the 6-character name of an existing semaphore (see Section 11.1.1.1).

status-record

The identifier of a variable of predefined record type EXC_STATUS that may receive an exception type and code. If you specify this parameter, the exception status, either success or error, that results from issuing the request is reported in this variable. Otherwise, an error causes the corresponding exception to be reported. The format of the exception record is described in Section 11.1.2.

Example

```
VAR
  Gate : SEMAPHORE_DESC;

[PRIORITY(10), STACK_SIZE(100), NAME ('CONSUM')] PROCESS Consumer;
BEGIN

  (* Wait on an unnamed semaphore. *)
  WAIT (DESC := Gate);

  (* Wait on a named semaphore. *)
  WAIT (NAME := 'GATE  ');

END; (* Process Consumer *)
```

Semantics

The WAIT procedure decrements the semaphore variable if its value is greater than 0 and returns control to the caller. If the semaphore value is 0, the WAIT procedure blocks the calling process until it is unblocked by a subsequent signal request. (See also the SIGNAL, COND_SIGNAL, and SIGNAL_ALL requests.)

This request is implemented through the WAIT$ kernel primitive.

Error Returns

See Section 11.2 for general information about error returns. The following exception codes may be returned:

ES$CDN	(type: SYSTEM_SERVICE)—Cannot specify both descriptor and name
ES$IST	(type: SYSTEM_SERVICE)—Illegal structure descriptor; no such semaphore exists
ES$MDN	(type: SYSTEM_SERVICE)—Must specify descriptor or name

13.19 WAIT_ANY

MACRO equivalent: WAIA$

The WAIT_ANY function implements a complex form of the wait on semaphore operation; see the WAIT and SIGNAL requests for a description of the basic wait and signal operations. WAIT_ANY performs the basic wait operation on the logical OR of several binary or counting semaphores, with an optional timeout feature. That is, WAIT_ANY permits the calling process to test for and, if necessary, wait on a signal on any one of a set of binary or counting semaphores. Up to four such semaphores can be specified in the request. If none of the specified semaphores can be decremented immediately, the calling process blocks until any one of those semaphores is signaled and can be decremented on behalf of the calling process.

Optionally, a WAIT_ANY operation can be terminated due to the expiration of a time interval specified in the request.

Thus, WAIT_ANY allows a process to synchronize with any of up to four events, each signaled by a separate process, for example. The function might also be used primarily for its timeout capability.

The function returns ordinal values from 0 to 5, indicating the results of the operation (see Semantics).

If a zero time period (immediate timeout) is specified in the request, WAIT_ANY provides a complex form of the COND_WAIT operation, which tests for a signaled semaphore but will not block the caller. See COND_WAIT for a description of the simple conditional wait operation.

Syntax

```
WAIT_ANY (  [[ SDB4 := sem-descriptor-4 ]]
            [[ SDB3 := sem-descriptor-3 ]]
            [[ SDB2 := sem-descriptor-2 ]]
            [[ SDB1 := sem-descriptor-1 ]]
            [[ TIMEOUT := timeout-interval ]]
            [[ STATUS := status-record ]] )
```

sem-descriptor-4
sem-descriptor-3
sem-descriptor-2
sem-descriptor-1

The identifier of a variable of predefined type SEMAPHORE_DESC that contains a semaphore's structure identifier. You can specify up to four binary or counting semaphores or a combination of both. The order in which you specify multiple semaphores determines the order in which they are initially tested for a signal. (That order can be critical under certain real-time conditions, as discussed under Semantics and Implementation Notes.) Each variable must have been previously initialized by a CREATE_BINARY_SEMAPHORE, CREATE_COUNTING_SEMAPHORE, or INIT_STRUCTURE_DESC request.

timeout-interval

The identifier of a variable of predefined type LONG_INTEGER that specifies the maximum time, in milliseconds, that the caller wishes to be blocked waiting for a signal. The value must be a positive integer from 0 to (2**31) –1. A value of 0 causes the request to time out immediately if none of the specified semaphores can be decremented without waiting.

That is, the calling process will never block if the specified time interval is 0. If you do not specify this parameter, the function assumes no timeout for the operation; the calling process may block indefinitely.

status-record

The identifier of a variable of predefined record type EXC_STATUS that may receive an exception type and code. If you specify this parameter, the exception status, either success or error, that results from issuing the request is reported in this variable. Otherwise, an error causes the corresponding exception to be reported. The format of the exception record is described in Section 11.1.2.

Restrictions

- The specified semaphore(s) must not be queue semaphore(s). (Binary and counting semaphores may be "intermixed" in the same request.)
- The timeout-interval value is limited to a 31-bit positive integer; that is, the sign bit of the high-order word must not be set. (The maximum valid value, in milliseconds, permits a timeout period of just over 24.89 days; see the SLEEP procedure for more detail.)
- If you wish to use fewer than four semaphore descriptor parameters, you must assign them, beginning with keyword SDB1. You may not assign keyword parameters with higher-numbered suffixes unless all keywords with lower-numbered suffixes are assigned. For example, if the parameter sequence specifies keyword SDB3, the sequence must also include keywords SDB2 and SDB1.

Example

```
%INCLUDE 'COMPLX.PAS'

VAR
  Line1, Line2, Line3 : SEMAPHORE_DESC;
  Which_one : COMPLEX_FUNC_VALUE;
  Timeout_val : LONG_INTEGER;

[PRIORITY(10), STACK_SIZE(100), NAME ('CONSUM')] PROCESS Consumer;
BEGIN

  (* Wait on three semaphores. *)
  Which_one := WAIT_ANY
    (SDB1 := Line1, SDB2 := Line2, SDB3 := Line3);

  (* Wait on two semaphores with a timeout. *)
  Timeout_val := 1000;
  Which_one := WAIT_ANY
    (SDB1 := Line1, SDB2 := Line2, TIMEOUT := Timeout_val);

END; (* Process Consumer *)
```

Semantics

The WAIT_ANY function tests each of the semaphores specified in the request for a gate-variable value greater than 0. That is, the function tests for a semaphore that is "open" and can be decremented. The semaphores are tested in the keyword order: SDB1 to SDB4.

The function returns the following values:

Value	Meaning
0	Request timed out
1	Request satisfied by SDB1
2	Request satisfied by SDB2
3	Request satisfied by SDB3
4	Request satisfied by SDB4
5	Error condition

If any of the semaphores are open at the time of the call, WAIT_ANY decrements the first open semaphore encountered and returns immediately to the caller, with a value between 1 and 4 that indicates which semaphore satisfied the request.

If none of the semaphores is open and either no timeout argument or a nonzero timeout value was supplied in the call, the function switches the calling process to the wait-active state, where the process is blocked on all the semaphores specified in the request.

If none of the semaphores is open and a zero timeout value was supplied in the call, the function returns immediately to the caller, with a zero value indicating a timeout. (The calling process thus never leaves the run state in the case of an immediate timeout.)

If an error occurs, the function returns a 5.

If the calling process switches to the wait-active state, the process is blocked from execution until it can be reactivated either by a signal on one of the blocking semaphores (see SIGNAL semantics) or by elapse of the specified timeout period, if any, before a signal occurs. When reactivated for either reason, the process is unblocked from all the semaphores and is switched to either the ready-active or the run state, depending on relative process priorities. If unblocked because of a signal, the function returns the ordinal value (from 1 to 4) of the semaphore that triggered the return, as described above. If unblocked because of a timeout, the function returns a 0.

This request is implemented through the WAIA$ kernel primitive.

Error Returns

See Section 11.2 for general information about error returns. The following exception code may be returned:

ES$IST	(type: SYSTEM_SERVICE)—Invalid structure descriptor; no such binary or counting semaphore exists

The request may return the following error, though not as a result of standard Pascal programming practice:

ES$IPM	(type: SYSTEM_SERVICE)—Illegal parameter; timeout value out of range

Implementation Notes

Since the initial test of the semaphores for a signal is performed in determinate order, the order in which multiple semaphores are identified in the call can be critical under certain real-time conditions. For example, assume that the relative frequency of signals is high for one of several binary or counting semaphores and that the "fast" semaphore is identified as being first, by being associated with keyword SDB1. In a series of calls to WAIT_ANY, that semaphore will be serviced far more often than the others, and the "slower" semaphores may seldom or never be tested and serviced.

Optimally, then, the semaphore with the highest expected signal rate should be assigned to the keyword that is tested last; the next highest as next to last, and so on, assuming that probable relative frequencies can be determined. Alternatively, the order in which the semaphores are identified could be rotated in successive calls so that at least N semaphores are guaranteed to be tested in N calls to WAIT_ANY. The correct or best strategy is application specific.

Chapter 14
Queue Semaphore Management Requests

This chapter describes the requests that operate on queue semaphore structures. These requests are the Pascal language interface to the services provided by the kernel's queue semaphore primitives. Table 14–1 lists the predeclared procedures and functions that implement these requests. They combine message-packet transmission with the synchronization features of signal and wait operations on counting semaphores.

A queue semaphore is a generalization of the counting semaphore and has a queue of elements associated with it in addition to the counter variable. (A standard MicroPower/Pascal element is called a packet.) Two distinct levels of queue semaphore operations are supplied, one built on the other. The basic signal operation adds a packet to the queue and increments the counter variable, implemented through the various forms of the PUT and SEND requests. The basic wait operation removes a packet from the queue and decrements the variable, implemented through the various forms of the GET and RECEIVE requests. If the queue is empty, the process must wait until an element can be removed. Thus, the value of the counter variable represents the number of elements on the queue.

The higher-level, more "automatic" operations are provided specifically for general processes in a mapped-memory environment. Those operations feature additional services that include copying of data by value and by reference between the sending process and the receiving process and signaling of a message-acknowledgment semaphore. Data transmission "by value" means that the message data is transmitted in the packet. Data transmission "by reference" means that a pointer to a referenced message buffer is transmitted in the packet.

You must include in your program or module the definitions of the routines that implement the requests described in this chapter before using them. See Appendix I for more information.

Table 14–1 lists the queue semaphore requests.

Table 14–1: Queue Semaphore Management Requests

General Requests	Function
DESTROY	Deletes a queue semaphore from the system and deallocates the memory space used by it.
GET_VALUE	Obtains a structure's value and type code.
INIT_STRUCTURE_DESC	Sets up a descriptor for efficient reference to a queue semaphore.

Low-Level Requests	Function
ALLOCATE_PACKET	Allocates a packet from the kernel's packet pool; if no packets are available, the calling process is blocked.
COND_ALLOCATE_PACKET	Conditional form of ALLOCATE_PACKET; performs the operation only if a packet is available.
COND_GET_PACKET	Conditional form of GET_PACKET; does not block the calling process if no packet is available.
COND_PUT_PACKET	Conditional form of PUT_PACKET; performs the operation only if a process is waiting on the semaphore.
CREATE_QUEUE_SEMAPHORE	Creates a queue semaphore in the system's common-memory area and sets up a descriptor for efficient reference to a queue semaphore.
CREATE_QUEUE_SEMAPHORE_P	Creates a queue semaphore in the system's common-memory area and sets up a descriptor for efficient reference to a queue semaphore by a procedure.
DEALLOCATE_PACKET	Returns a packet to the kernel's packet-allocation pool.
GET_PACKET	Tests specified queue semaphore for a packet of data: if one is available, the packet is detached from the queue, and a pointer to it is passed to the calling process; if no packet is available, the process is blocked.
GET_PACKET_ANY	An enhanced form of GET_PACKET that waits for a variable time interval on up to four queue semaphores.
PUT_PACKET	Places a packet of data on a queue; if any processes are waiting on that semaphore, the first process is unblocked.

Table 14–1 (Cont.): Queue Semaphore Management Requests

High-Level Requests	Function
COND_RECEIVE	Conditional form of RECEIVE; does not block the calling process if no packet is available.
COND_SEND	Conditional form of SEND; performs the operation only if a process is waiting on the semaphore.
RECEIVE	Waits on a queue semaphore for an available packet, then copies a message from the sender's buffer or up to 34 bytes of message data, or both, from the packet into the caller's buffers.
RECEIVE_ANY	An enhanced form of RECEIVE that waits for a variable time interval on up to four queue semaphores.
SEND	Copies up to 34 bytes of message data or a reference to a message buffer, or both, into a packet and places the packet on a queue.

High-Level Requests with Message Reply Capability	Function
COND_RECEIVE_ACK	Conditional form of RECEIVE_ACK; does not block the calling process if no packet is available.
COND_SEND_ACK	Conditional form of SEND_ACK; performs the operation only if a process is waiting on the queue semaphore.
RECEIVE_ACK	Waits on a queue semaphore for an available packet, then copies a message from the sender's buffer to the caller's buffer and optionally signals a reply semaphore identified by the sender.
RECEIVE_ANY_ACK	An enhanced form of RECEIVE_ACK that waits for a variable time interval on up to four queue semaphores.
SEND_ACK	Copies a reference to a message and an optional reply semaphore identifier into a packet and places the packet on a queue.

14.1 Data Access Features of Processes

In a mapped-memory environment, a process must have special mapping attributes (driver mapping or privileged mapping) to use fully the lower-level queue semaphore requests (PUT_PACKET, GET_PACKET, and their conditional forms). Since those requests directly access the contents of a packet and since packets reside in kernel data space, the process must be mapped to that space to access (write into or read from) the packet. Therefore, a program containing a process that uses those requests must be declared with the PRIVILEGED or DRIVER attribute (see Section 10.1.2).

Processes with either general mapping or DEV_ACCESS mapping do not have direct access to the contents of a packet and therefore may not store data directly into a packet. If such a process needs to access the contents of a packet rather than pass one along that the process has acquired by means of another request, the process must use a form of the SEND and RECEIVE requests. Those requests provide a packet-creation and data-copying service in addition to the functionality of the PUT_PACKET/GET_PACKET combination.

Note

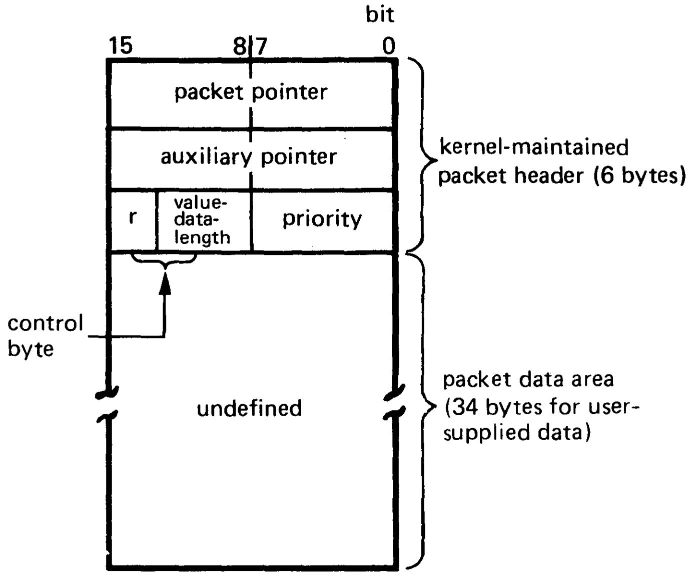
Since packets exist in kernel data space, a general-mapping process attempting to access a packet will obtain unpredictable results. If the packet address is not also a valid virtual address in the process's address space, the kernel will generate a memory-management exception (ES$MMU). No exception will occur if the packet address is a valid virtual address in the process's space, but the process will then obtain invalid data.

For further information about mapping strategy, refer to Chapter 2 of the *MicroPower/Pascal Run-Time Services Manual.*

14.2 General Packet Structure for Send/Receive Requests

A packet is a standard fixed-length data structure that the kernel allocates from a special system-memory pool. A packet's overall size is 40 bytes (see Figure 14–1), including the header; that of the undefined, arbitrarily usable portion of the packet is 34 bytes. (Those standard sizes are provided in the MicroPower/Pascal software as distributed by DIGITAL.)

Figure 14–1: General Packet Format for Send/Receive Requests



bit
15
8
7
0
packet pointer
auxiliary pointer
r
value-data-length
priority
kernel-maintained packet header (6 bytes)
control byte
undefined
packet data area (34 bytes for user-supplied data)

MLO–563–87

Descriptions of the components of the packet format shown in Figure 14–1 follow:

packet pointer

The kernel-maintained pointer to the next packet in the queue; for kernel use only.

auxiliary pointer

For DIGITAL use only.

r

The reference data flag, provided by the various forms of the SEND request to indicate that a pointer to a data buffer is in the packet.

0 = no data by reference specified
1 = data by reference specified

value-data-length

The value data byte count, provided by the various forms of the SEND request.

priority

The packet-priority value, provided by the various forms of the SEND request.

A process obtains a packet from the kernel by issuing either an ALLOCATE_PACKET or a COND_ALLOCATE_PACKET request. (The undefined portion of the packet returned by the kernel is not initialized.)

A process issues a DEALLOCATE_PACKET request to return a packet to the system's free-element pool.

Refer to Figures 14–2 and 14–3 for specific layouts of packets used with the SEND, COND_SEND, SEND_ACK, and COND_SEND_ACK requests.

14.3 ALLOCATE_PACKET

MACRO equivalent: ALPK$

The ALLOCATE_PACKET procedure obtains a message packet (standard queue element) from the kernel's free-packet pool. If available, a free packet is logically removed from the free-packet pool, and a pointer to the packet is returned to the caller. If all packets are in use at the time of the call, the calling process is blocked until the request can be satisfied. (If several processes are concurrently waiting for packet allocation, the requests are satisfied according to process priority as packets are returned to the pool.)

The procedure permits the caller to obtain a packet pointer for use with the GET_PACKET, GET_PACKET_ANY, COND_GET_PACKET, PUT_PACKET, and COND_PUT_PACKET requests. This procedure is for use by processes with the PRIVILEGED or DRIVER attributes or processes that reside in an unmapped-memory environment (see Sections 10.1.2 and 14.1 for further information).

The COND_ALLOCATE_PACKET function permits a process to request packet allocation without blocking if no packets are free.

The DEALLOCATE_PACKET procedure, the complement of ALLOCATE_PACKET, lets a process deallocate a message packet.

Syntax

ALLOCATE_PACKET (PACKET_PTR := pointer)

pointer

The identifier of a variable of predefined type QUEUE_PTR that will receive a pointer to the packet.

Example

```
VAR
  Pack : QUEUE_PTR;

[PRIORITY(10), STACK_SIZE(100), NAME ('P1    ')] PROCESS P1;
BEGIN

  (* Allocate a packet. *)
  ALLOCATE_PACKET (PACKET_PTR := Pack);

END; (* Process P1 *)
```

Semantics

The ALLOCATE_PACKET procedure tests the free-packet pool for a free packet. If the pool contains at least one packet, the procedure logically removes a packet from the pool and returns the address of that packet in the variable specified by the pointer parameter.

If no packets are free, the request blocks the calling process on a semaphore ($KPSEM) associated with the free-packet pool and calls the scheduler. The process remains on the semaphore's waiting process list, in priority order relative to other processes that may also be waiting, until enough packets have been freed to permit allocation. (See the DEALLOCATE_PACKET procedure.)

This request is implemented through the ALPK$ kernel primitive.

Error Returns

None

14.4 COND_ALLOCATE_PACKET

MACRO equivalent: ALPC$

The COND_ALLOCATE_PACKET function (the conditional, or nonblocking, form of the ALLOCATE_PACKET request) obtains a message packet from the kernel's free-packet pool, if one is available, and returns to the caller with a Boolean TRUE value. If all packets are in use at the time of the call, the function returns control to the caller immediately, with a Boolean FALSE value.

This function permits the caller to obtain a packet pointer for use with the GET_PACKET, GET_PACKET_ANY, COND_GET_PACKET, PUT_PACKET, and COND_PUT_PACKET requests without blocking if the request cannot be satisfied. (Compare with the unconditional form, ALLOCATE_PACKET.) This function is for use by processes with the PRIVILEGED or DRIVER attributes or processes that reside in an unmapped-memory environment (see Sections 10.1.2 and 14.1 for more information).

The DEALLOCATE_PACKET procedure lets a process deallocate a message packet.

Syntax

COND_ALLOCATE_PACKET (PACKET_PTR := pointer)

pointer

The identifier of a variable of predefined type QUEUE_PTR that will receive a pointer to the packet.

Example

```
VAR
  Pack : QUEUE_PTR;
  Access : SEMAPHORE_DESC;

[PRIORITY(10), STACK_SIZE(100), NAME ('P1    ')] PROCESS P1;
BEGIN

  (* Conditionally allocate a packet.  If the allocation fails then
     wait for someone to return one. *)
  IF NOT COND_ALLOCATE_PACKET (PACKET_PTR := Pack)
   THEN WAIT (DESC := Access);

END; (* Process P1 *)
```

Semantics

The COND_ALLOCATE_PACKET function tests the free-packet pool for a free packet. If the pool contains at least one packet, the function does the following:

1. Logically removes a packet from the pool.
2. Returns the address of that packet in the variable specified by the pointer parameter.
3. Returns a Boolean TRUE value.

If no packets are free, the function returns immediately to the calling process, with a Boolean FALSE value.

This request is implemented through the ALPC$ kernel primitive.

Error Returns

None

14.5 COND_GET_PACKET

MACRO equivalent: WAQC$

The COND_GET_PACKET function implements the conditional, or nonblocking, form of the GET_PACKET request. This function permits the calling process to receive a signal from another process that a data packet is available, without being blocked on the semaphore if it has not been signaled yet.

This function is for use by processes with the PRIVILEGED or DRIVER attributes or processes that reside in an unmapped-memory environment (see Sections 10.1.2 and 14.1 for more information).

The COND_PUT_PACKET and PUT_PACKET requests allow a process to place a packet in the queue of a queue semaphore.

Syntax

```
COND_GET_PACKET (  PACKET_PTR := pointer
                   { DESC := queue-sem-descriptor }
                   { NAME := queue-sem-name       }
                   [[ STATUS := status-record ]] )
```

pointer

The identifier of a variable of predefined type QUEUE_PTR that will receive a pointer to the packet being obtained.

queue-sem-descriptor

The identifier of a variable of predefined type QUEUE_SEMAPHORE_DESC that contains the semaphore's structure identifier. The variable must have been previously initialized by a CREATE_QUEUE_SEMAPHORE or an INIT_STRUCTURE_DESC request.

queue-sem-name

A character-string constant or variable of predefined type NAME_STR that specifies the 6-character name of an existing semaphore (see Section 11.1.1.1).

status-record

The identifier of a variable of predefined record type EXC_STATUS that may receive an exception type and code. If you specify this parameter, the exception status, either success or error, that results from issuing the request is reported in this variable. Otherwise, an error causes the corresponding exception to be reported. The format of the exception record is described in Section 11.1.2.

Example

```
VAR
  Queue_1 : QUEUE_SEMAPHORE_DESC;
  Pack : QUEUE_PTR;

[PRIORITY(10), STACK_SIZE(100), NAME ('CONSUM')] PROCESS Consumer;
BEGIN
```

```
  (* Conditionally get a packet via an unnamed queue semaphore. *)
  IF COND_GET_PACKET (PACKET_PTR := Pack,
                      DESC := Queue_1)
   THEN (* Use the data. *);

  (* Conditionally get a packet via a named queue semaphore. *)
  IF COND_GET_PACKET (PACKET_PTR := Pack,
                      NAME := 'QUEUE1')
   THEN (* Use the data. *);

END; (* Process Consumer *)
```

Semantics

The COND_GET_PACKET function tests the specified queue semaphore for an available packet. If at least one packet is in the semaphore's queue, the function does the following:

1. Removes the first available packet from the queue.
2. Decrements the specified queue semaphore.
3. Places a pointer to the packet in the variable specified by the pointer parameter.
4. Returns a Boolean TRUE value to the caller.

If no packets are on the semaphore's queue, the function returns to the caller, with a Boolean FALSE value.

This request is implemented through the WAQC$ kernel primitive.

Error Returns

See Section 11.2 for general information about error returns. The following exception codes may be returned:

ES$CDN	(type: SYSTEM_SERVICE)—Cannot specify both descriptor and name
ES$IST	(type: SYSTEM_SERVICE)—Illegal structure descriptor; no such queue semaphore exists
ES$MDN	(type: SYSTEM_SERVICE)—Must specify descriptor or name

Applications

The COND_GET_PACKET request can be used to determine if a particular event has occurred without the possibility of the process blocking. Such a request is useful in time-critical situations when the caller cannot afford to block waiting for an event, and when strict synchronization is not required.

14.6 COND_PUT_PACKET

MACRO equivalent: SGQC$

The COND_PUT_PACKET function implements the conditional, or nonblocking, form of the PUT_PACKET request. This function permits the calling process to pass a data packet to another process, but only if the other process is already waiting for the packet. (Compare with the unconditional form, PUT_PACKET.)

This function is for use by processes with the PRIVILEGED or DRIVER attributes or processes that reside in an unmapped-memory environment (see Sections 10.1.2 and 14.1 for more information).

The COND_GET_PACKET and GET_PACKET requests let a process obtain a packet from a queue semaphore.

Syntax

```
COND_PUT_PACKET (  PACKET_PTR := pointer
                   { DESC := queue-sem-descriptor }
                   { NAME := queue-sem-name       }
                   [[ STATUS := status-record ]] )
```

pointer

The identifier of a variable of predefined type QUEUE_PTR that contains a pointer to the packet being sent.

queue-sem-descriptor

The identifier of a variable of predefined type QUEUE_SEMAPHORE_DESC that contains the semaphore structure identifier. The variable must have been previously initialized by a CREATE_QUEUE_SEMAPHORE or an INIT_STRUCTURE_DESC request.

queue-sem-name

A character-string constant or variable of predefined type NAME_STR that specifies the 6-character name of an existing semaphore (see Section 11.1.1.1).

status-record

The identifier of a variable of predefined record type EXC_STATUS that may receive an exception type and code. If you specify this parameter, the exception status, either success or error, that results from issuing the request is reported in this variable. Otherwise, an error causes the corresponding exception to be reported. The format of the exception record is described in Section 11.1.2.

Example

```
VAR
  Queue_1 : QUEUE_SEMAPHORE_DESC;
  Pack : QUEUE_PTR;
  Failures : UNSIGNED;

[PRIORITY(10), STACK_SIZE(100), NAME ('PRODUC')] PROCESS Producer;
BEGIN
```

```
  (* Conditionally  send a packet via  an unnamed queue semaphore
     while keeping count of failures. *)
  IF NOT COND_PUT_PACKET (PACKET_PTR := Pack,
                          DESC := Queue_1)
   THEN Failures := Failures + 1;

  (* Conditionally send a  packet  via  a  named  queue semaphore
     while keeping count of failures. *)
  IF NOT COND_PUT_PACKET (PACKET_PTR := Pack,
                          NAME := 'QUEUE1')
   THEN Failures := Failures + 1;

END; (* Process Producer *)
```

Semantics

The COND_PUT_PACKET function tests the specified semaphore for waiting processes. If at least one process is waiting on the semaphore, the function does the following:

1. Unblocks the first waiting process.
2. Associates the passed packet pointer with that process as its wait-return value.
3. Calls the scheduler.

This sequence may cause the calling process to be preempted: to lose control of the CPU. When the calling process eventually gains control, the function returns a Boolean TRUE value to indicate a successful operation.

If no process is waiting on the semaphore, the function returns control immediately to the caller, with a Boolean FALSE value to indicate an unsuccessful operation.

This request is implemented through the SGQC$ kernel primitive.

Error Returns

See Section 11.2 for general information about error returns. The following exception codes may be returned:

ES$CDN	(type: SYSTEM_SERVICE)—Cannot specify both descriptor and name
ES$IST	(type: SYSTEM_SERVICE)—Illegal structure descriptor; no such queue semaphore exists
ES$MDN	(type: SYSTEM_SERVICE)—Must specify descriptor or name

The request may return the following error, though not as a result of standard Pascal programming practice:

ES$IAD	(type: SYSTEM_SERVICE)—Invalid address; pointer to buffer or structure is odd or not in user address space

Applications

This request permits a process to send a record (packet) to any of several queue semaphores, based on the condition that another process is waiting for the record. For example, suppose that a process wishes to send an output request contained in a packet to any one of three output-service processes associated with separate queue semaphores. The COND_PUT_PACKET request lets the submitting process test each semaphore for an output process that is ready to service the request.

14.7 COND_RECEIVE

MACRO equivalent: RCVC$

The COND_RECEIVE function implements the conditional, or nonblocking, form of the RECEIVE request. This function tests the specified queue semaphore for an available packet. If a packet is available, COND_RECEIVE obtains the packet's pointer, copies data from or through it to the caller's buffer space, and returns a Boolean TRUE value to the caller. If no packet is available, the function returns control immediately to the caller, with a Boolean FALSE value instead of blocking the process on the semaphore, as with the RECEIVE request. The packet format expected by COND_RECEIVE is the same as that produced by the SEND and COND_SEND requests, as described in Figure 14–2.

The message-reception features of COND_RECEIVE are identical to those provided by the RECEIVE request, that is, the copying of messages sent either by value or by reference. The only functional difference between the two requests is the unconditional wait performed by RECEIVE versus the conditional wait performed by COND_RECEIVE.

Note

The message-by-reference feature must be used with caution concerning the length of the message. Once the message-copying operation begins, no other process can gain control until the entire message is copied, because of the indivisible nature of kernel primitive operations. Thus, transmission of long messages can seriously affect the servicing of interrupts by increasing interrupt latency throughout the system.

COND_RECEIVE is intended for use by processes with general or DEV_ACCESS mapping; such processes cannot access a packet directly in a mapped-memory environment (see Section 10.1.2). This function allows any process, regardless of mapping type, to obtain a packet of data from another process.

The SEND and COND_SEND requests permit a process to transmit data through a packet.

Syntax

```
COND_RECEIVE ( ⟦ REF_DATA := reference-data-id
               REF_LENGTH := reference-data-length ⟧
               ⟦ VAL_DATA := value-data-id
               VAL_LENGTH := value-data-length ⟧
               ⟦ RET_INFO := information-record ⟧
               { DESC := queue-sem-descriptor }
               { NAME := queue-sem-name       }
               ⟦ STATUS := status-record ⟧ )
```

reference-data-id

The identifier of the variable (buffer) that will contain the data being received by reference. This parameter is significant only if the reference-data-length parameter is nonzero.

reference-data-length

A constant or the identifier of a variable of predefined type REF_DATA_LEN that specifies the length, in bytes, of the message buffer (reference-data-id parameter) containing the data to be sent by reference. This parameter limits the amount of data to be copied from the sender's buffer. If the value is 0 and if a message by reference exists in the packet, the message is not copied; the reference is passed to the receiver in the record specified by the information-record parameter. The maximum value is 8128 bytes. The default value is 0.

value-data-id

The identifier of the variable (buffer) that will contain the data being sent by value. The contents of this buffer are copied directly from the packet. This parameter is significant only if the value-data-length parameter is nonzero.

value-data-length

A constant or the identifier of a variable of predefined type VAL_DATA_LEN that specifies the length, in bytes, of the buffer identified by the value-data-id parameter. This parameter limits the amount of data to be copied from the packet. The value of this parameter can be from 0 to 34 inclusively; the default value is 0.

information-record

The identifier of a variable of predefined type INFO_BLOCK that may receive status information about the operation.

queue-sem-descriptor

The identifier of a variable of predefined type QUEUE_SEMAPHORE_DESC that contains the semaphore's structure identifier. The variable must have been previously initialized by a CREATE_QUEUE_SEMAPHORE or an INIT_STRUCTURE_DESC request.

queue-sem-name

A character-string constant or the identifier of a variable of predefined type NAME_STR that specifies the 6-character name of an existing semaphore (see Section 11.1.1.1).

status-record

The identifier of a variable of predefined record type EXC_STATUS that may receive an exception type and code. If you specify this parameter, the exception status, either success or error, that results from issuing the request is reported in this variable. Otherwise, an error causes the corresponding exception to be reported. The format of the exception record is described in Section 11.1.2.

Format of Information Record

The information returned to the caller in the variable specified by the information-record parameter is a record of predefined type INFO_BLOCK as follows:

```
INFO_BLOCK = PACKED RECORD
                PRIORITY       : [BYTE] PRIORITY_RANGE;
                VAL_XMIT_LEN   : [BYTE] VAL_DATA_LEN;
                ADDRESS        : PHYSICAL_ADDRESS;
                REF_XMIT_LEN   : [WORD] REF_DATA_LEN;
             END;
```

PRIORITY

The priority value that was assigned to the packet by the send operation.

VAL_XMIT_LEN

The number of bytes that were sent by value. This value may be greater than the number of bytes received, which is limited by the value-data-length parameter. A 0 indicates that no data by value was sent.

ADDRESS

A record of predefined type PHYSICAL_ADDRESS that contains the physical address of the sender's message-by-reference buffer, if any. The format of the record is:

```
PHYSICAL_ADDRESS = PACKED RECORD
                     ADDRESS     : UNSIGNED;
                     PAR_VALUE   : UNSIGNED;
                   END;
```

ADDRESS

The address within the sender's address space of the message-by-reference buffer. This return value is valid only if the REF_XMIT_LEN value is nonzero; otherwise, the contents of this word are unpredictable.

PAR_VALUE

The value of the page address register (PAR) that maps the sender's message-by-reference buffer, if any. This return value is valid only in a mapped-memory environment when the REF_XMIT_LEN value is nonzero; otherwise, the contents of this word are unpredictable.

REF_XMIT_LEN

The number of bytes that were sent by reference. This value may be greater than the number of bytes received, which is limited by the reference-data-length parameter. A 0 indicates that no data by reference was sent.

Restrictions

- The maximum value for the reference-data-length parameter is 8128.
- A total of 34 bytes is available in a packet for message data, a message reference, or both. A reference occupies three words in the packet and, if included, reduces the space available for data by value to 28 bytes.

Example

```
VAR
  Queue_1 : QUEUE_SEMAPHORE_DESC;
  Buffer : PACKED ARRAY [0..511] OF CHAR;
  Length : 0..512;
  Info : INFO_BLOCK;

[PRIORITY(10), STACK_SIZE(100), NAME ('CONSUM')] PROCESS Consumer;
BEGIN

  (* Conditionally receive data by reference. *)
  IF COND_RECEIVE (REF_DATA := Buffer,
                   REF_LENGTH := Length,
                   RET_INFO := Info,
                   DESC := Queue_1)
   THEN (* Use the data. *);
  (* Conditionally receive data by value. *)
  IF COND_RECEIVE (VAL_DATA := Buffer,
                   VAL_LENGTH := Length,
                   RET_INFO := Info,
                   DESC := Queue_1)
   THEN (* Use the data. *)
END; (* Process Consumer *)
```

Semantics

The COND_RECEIVE function tests the specified queue semaphore for an available packet. If a packet is available, the function removes the packet from the semaphore's queue and performs the following actions as governed by the parameter specified in the call:

1. Copies data sent by value, if any, from the packet in system space to the caller's value-data-id buffer area. The number of bytes copied is the lesser of the value-data-length parameter value and the number of bytes sent by value.
2. Copies data sent by reference, if any, from the sender's message buffer to the caller's reference-data-id buffer area. The number of bytes copied is the lesser of the reference-data-length parameter value and the REF_XMIT_LENGTH field returned in the variable specified by the information-record parameter.
3. Copies the priority of the packet and the number of bytes sent by value from the packet header to the PRIORITY and VAL_XMIT_LEN fields of the receiver's information-record area.
4. Copies the message reference, if any, contained in the packet to the corresponding three words of the receiver's information-record area.
5. Zeros the REF_XMIT_LEN field of the receiver's information-record area if the packet contains no message reference.
6. Deallocates the packet, returning it to the system's free-element pool for reuse.
7. Returns control to the caller, with a Boolean TRUE value.

If no packet is queued on the specified semaphore at the time of the call, the function returns immediately to the caller, with a Boolean FALSE value to indicate that the conditional receive operation was unsuccessful.

The packet format expected by the COND_RECEIVE request is described in Figure 14–2.

This request is implemented through the RCVC$ kernel primitive.

Error Returns

See Section 11.2 for general information about error returns. The following exception codes may be returned:

ES$CDN	(type: SYSTEM_SERVICE)—Cannot specify both descriptor and name
ES$IST	(type: SYSTEM_SERVICE)—Illegal structure descriptor; no such queue semaphore exists
ES$MDN	(type: SYSTEM_SERVICE)—Must specify descriptor or name

The request may return the following error, though not as a result of standard Pascal programming practice:

ES$IAD	(type: SYSTEM_SERVICE)—Invalid address; pointer to buffer or rtnptr is not a word-address (even) value

14.8 COND_RECEIVE_ACK

MACRO equivalent: RCVC$

The COND_RECEIVE_ACK function implements the conditional, or nonblocking, form of the RECEIVE_ACK request. This function tests the specified queue semaphore for an available packet. If a packet is available, the function performs the following operations:

- Copies any referenced data from the sender's buffer to the receiver's buffer
- Signals the reply semaphore automatically or passes the structure identifier of a reply semaphore to the receiver for manual signaling if the sender specified a reply semaphore
- Returns a Boolean TRUE value

If no packet is available, the function returns control immediately to the caller, with a Boolean FALSE value.

The packet format expected by COND_RECEIVE_ACK is the same as that produced by the SEND_ACK and COND_SEND_ACK requests as discussed in Figure 14-3.

Note

The message-by-reference feature must be used with caution concerning the length of the message. Once the message-copying operation begins, no other process can gain control until the entire message is copied, because of the indivisible nature of kernel primitive operations. Thus, transmission of long messages can seriously affect the servicing of interrupts by increasing interrupt latency throughout the system.

COND_RECEIVE_ACK is intended for use by processes with general or DEV_ACCESS mapping; such processes cannot access a packet directly in a mapped-memory environment (see Section 10.1.2). This function permits a process, regardless of mapping type, to obtain data by reference and a reply semaphore from another process through a packet.

The SEND_ACK and COND_SEND_ACK requests permit a process to transmit data by reference and a reply semaphore through a packet.

Syntax

```
COND_RECEIVE_ACK ( [[ REF_DATA := reference-data-id
                      REF_LENGTH := reference-data-length ]]
                   [[ REPLY_DESC := reply-sem-descriptor ]]
                   REC_LENGTH := ref-xmit-length
                   { DESC := queue-sem-descriptor }
                   { NAME := queue-sem-name       }
                   [[ STATUS := status-record ]] )
```

reference-data-id

The identifier of the variable (buffer) that may contain the data being received by reference. This parameter is significant only if the reference-data-length parameter is nonzero.

reference-data-length

A constant or the identifier of a variable of predefined type REF_DATA_LEN that specifies the length, in bytes, of the message buffer (reference-data-id parameter) containing the data to be sent by reference. This parameter limits the amount of data to be copied from the sender's buffer. If the value is 0 and if a message by reference exists in the packet, the message is not copied. The maximum value is 8128 bytes.

reply-sem-descriptor
The identifier of a variable of predefined type SEMAPHORE_DESC that will receive the structure identifier of the binary or counting reply semaphore provided by the sender. If you do not specify this parameter, the procedure automatically signals the reply semaphore when the receive operation is complete. Otherwise, the reply semaphore may be signaled manually at the receiver's discretion.

ref-xmit-length
The identifier of a variable of predefined type REF_DATA_LEN that will receive a value that is the number of bytes of data sent by reference. This value may be greater than the number of bytes received, which is limited by the reference-data-length parameter. Zero indicates that no data by reference was sent.

queue-sem-descriptor
The identifier of a variable of predefined type QUEUE_SEMAPHORE_DESC that contains the semaphore's structure identifier. The variable must have been previously initialized by a CREATE_QUEUE_SEMAPHORE or an INIT_STRUCTURE_DESC request.

queue-sem-name
A character-string constant or the identifier of a variable of predefined type NAME_STR that specifies the 6-character name of an existing semaphore (see Section 11.1.1.1).

status-record
The identifier of a variable of predefined record type EXC_STATUS that may receive an exception type and code. If you specify this parameter, the exception status, either success or error, that results from issuing the request is reported in this variable. Otherwise, an error causes the corresponding exception to be reported. The format of the exception record is described in Section 11.1.2.

Example

```
VAR
  Queue_1 : QUEUE_SEMAPHORE_DESC;
  Buffer : PACKED ARRAY [0..511] OF CHAR;
  Length : 0..512;
  Received : REF_DATA_LEN;
  Reply : SEMAPHORE_DESC;

[PRIORITY(10), STACK_SIZE(100), NAME ('CONSUM')] PROCESS Consumer;
BEGIN

  (* Conditionally receive data with acknowledgment. *)
  IF COND_RECEIVE_ACK (REF_DATA := Buffer,
                     REF_LENGTH := Length,
                     REPLY_DESC := Reply,
                     REC_LENGTH := Received,
                     DESC := Queue_1)
   THEN SIGNAL (DESC := Reply);

END; (* Process Consumer *)
```

Restriction

The maximum value for the reference-data-length parameter is 8128.

Semantics

The COND_RECEIVE_ACK function decrements the specified queue semaphore and tests for an available packet. If at least one packet is on the semaphore's queue, the function removes the first available packet from the queue and performs the following operations:

1. Copies data sent by reference, if any, from the sender's message buffer to the caller's reference-data-id buffer area. The number of bytes copied is the lesser of the reference-data-length parameter value and the ref-xmit-length parameter.
2. Returns a 0 in the variable specified by the ref-xmit-length parameter if the message reference in the packet contained no data.
3. Signals the reply semaphore automatically if the receiver did not specify the reply-sem-descriptor parameter and the sender passed a reply semaphore structure identifier in the packet, or returns the reply semaphore structure identifier if the reply-sem-descriptor parameter was specified and the sender passed a reply semaphore structure identifier in the packet.
4. Deallocates the packet, returning it to the system's free-element pool for reuse.
5. Returns control to the caller, with a Boolean TRUE value.

If no packets are on the semaphore's queue, the function returns immediately to the caller, with a Boolean FALSE value to indicate that the conditional operation was unsuccessful.

The packet format expected by the COND_RECEIVE_ACK request is described in Figure 14–3.

This request is implemented through the RCVC$ kernel primitive.

Error Returns

See Section 11.2 for general information about error returns. The following exception codes may be returned:

ES$CDN	(type: SYSTEM_SERVICE)—Cannot specify both descriptor and name
ES$IST	(type: SYSTEM_SERVICE)—Illegal structure descriptor; reply semaphore or queue semaphore does not exist
ES$MDN	(type: SYSTEM_SERVICE)—Must specify descriptor or name
ES$RDE	(type: SYSTEM_SERVICE)—Reply descriptor expected

The request may return the following error, though not as a result of standard Pascal programming practice:

ES$IAD	(type: SYSTEM_SERVICE)—Invalid address; pointer to buffer or rtnptr is not a word-address (even) value

14.9 COND_SEND

MACRO equivalent: SNDC$

The COND_SEND function implements a conditional form of the SEND request. If the function finds a process waiting (blocked) on the specified queue semaphore, this function performs the following operations:

- Allocates a packet in system space
- Copies user data into the packet
- Signals the queue semaphore in the same way as the PUT_PACKET request
- Returns a Boolean TRUE value to the caller

If no process is waiting on the semaphore, the function returns control immediately to the caller with a Boolean FALSE value.

A packet constructed by COND_SEND has the same format as one constructed by the SEND request.

The message-transmission characteristics and packet structure used by COND_SEND are identical to those provided by SEND, that is, the sending of messages by value or by reference. The only functional difference between the two requests is the unconditional signal performed by SEND versus the conditional signal performed by COND_SEND.

Note

> The message-by-reference feature must be used with caution concerning the length of the message. Once the message-copying operation begins, no other process can gain control until the entire message is copied, because of the indivisible nature of primitive operations. Thus, transmission of long messages can seriously affect the servicing of interrupts by increasing interrupt latency throughout the system.

COND_SEND is intended for use by processes with general or DEV_ACCESS mapping; such processes cannot access a packet directly in a mapped-memory environment (see Section 10.1.2). This function permits a process, regardless of mapping type, to transmit data to another process through a packet.

The RECEIVE and COND_RECEIVE requests permit a process to receive data sent through a packet.

Syntax

```
COND_SEND ( [[ REF_DATA := reference-data-id
              REF_LENGTH := reference-data-length ]]
            [[ VAL_DATA := value-data-id
              VAL_LENGTH := value-data-length ]]
            [[ PRIORITY := packet-priority ]]
            { DESC := queue-sem-descriptor }
            { NAME := queue-sem-name       }
            [[ STATUS := status-record ]] )
```

reference-data-id

The identifier of the variable (buffer) that contains the data to be sent by reference. The address of this variable is converted to a physical address and is placed in the packet with the reference-data-length parameter value. This parameter is significant only if the reference-data-length parameter is nonzero.

reference-data-length

A constant or the identifier of a variable of predefined type REF_DATA_LEN that specifies the length, in bytes, of the message buffer (reference-data-id parameter) containing the data to be sent by reference. The maximum value is 8128 bytes.

value-data-id

The identifier of the variable (buffer) that contains the data to be sent by value. The contents of this buffer are copied into the packet directly. This parameter is significant only if the value-data-length parameter is nonzero.

value-data-length

A constant or the identifier of a variable of predefined type VAL_DATA_LEN that specifies the number of bytes to be transmitted by value. The maximum is 34 if no message is sent by reference (that is, if reference-data-length = 0) or 28 if a message reference is specified. The default value is 0.

packet-priority

A constant or the identifier of a variable of predefined type PRIORITY_RANGE that specifies the priority value (0 to 255) to be assigned to the packet. This value affects the order in which the packet is queued on a semaphore having a priority-ordered packet queue (see the CREATE_QUEUE_SEMAPHORE request). The default priority value is 1.

queue-sem-descriptor

The identifier of a variable of predefined type QUEUE_SEMAPHORE_DESC that contains the semaphore's structure identifier. The variable must have been previously initialized by a CREATE_QUEUE_SEMAPHORE or an INIT_STRUCTURE_DESC request.

queue-sem-name

A character-string constant or the identifier of a variable of predefined type NAME_STR that specifies the 6-character name of an existing semaphore (see Section 11.1.1.1).

status-record

The identifier of a variable of predefined record type EXC_STATUS that may receive an exception type and code. If you specify this parameter, the exception status, either success or error, that results from issuing the request is reported in this variable. Otherwise, an error causes the corresponding exception to be reported. The format of the exception record is described in Section 11.1.2.

You must specify at least one set of parameters; either value or reference.

Restrictions

- The maximum value for the reference-data-length parameter is 8128.
- A total of 34 bytes is available in a packet for message data, a message reference, or both. A reference occupies three words in the packet and, if included, reduces the space available for data by value to 28 bytes.

Example

```
VAR
  Queue_1 : QUEUE_SEMAPHORE_DESC;
  Buffer : PACKED ARRAY [0..511] OF CHAR;
  Failures : UNSIGNED;

[PRIORITY(10), STACK_SIZE(100), NAME ('PRODUC')] PROCESS Producer;
BEGIN

  (* Conditionally send data by reference while keeping count of failures. *)
  IF NOT COND_SEND (REF_DATA := Buffer,
                    REF_LENGTH := 512,
                    DESC := Queue_1)
   THEN Failures := Failures + 1;

  (* Conditionally send data by value while keeping count of failures. *)
  IF NOT COND_SEND (VAL_DATA := Buffer,
                    VAL_LENGTH := 12,
                    DESC := Queue_1)
   THEN Failures := Failures + 1;

END; (* Process Producer *)
```

Semantics

The COND_SEND function tests the specified queue semaphore for a waiting process. If a process is waiting, the function performs the following actions prior to signaling the semaphore:

1. Obtains a packet from the system's free-element pool and writes the specified priority value into the packet header.
2. Constructs a control byte based on the value-data-length and reference-data-length parameters and places it in the packet header for subsequent use by the RECEIVE and COND_RECEIVE requests.
3. Copies the data, if any, to be transmitted by value from the buffer in user space to the packet in system space.
4. Constructs a physical address from the address (reference-data-id) of the message to be sent by reference, if any. This physical address is placed in the packet with the message length. A physical address consists of two words. The value of the first word is the virtual address; the value of the second word is the content of the user-mode PAR associated with that virtual address.

If at least one process is waiting, COND_SEND unblocks the first waiting process, associates the packet with that process (as its wait-return value), and calls the scheduler, which may cause the calling process to be preempted: to lose control of the CPU. On eventual return to the caller, the function returns a Boolean TRUE value, indicating a successful operation.

If no process is waiting on the semaphore, the function returns immediately to the caller, with a Boolean FALSE value to indicate that the send operation was not performed.

A packet constructed by COND_SEND has the same format as one constructed by SEND, as described in Figure 14–2.

This request is implemented through the SNDC$ kernel primitive.

Error Returns

See Section 11.2 for general information about error returns. The following exception codes may be returned:

ES$CDN	(type: SYSTEM_SERVICE)—Cannot specify both descriptor and name
ES$IST	(type: SYSTEM_SERVICE)—Illegal structure descriptor; no such queue semaphore exists
ES$MDN	(type: SYSTEM_SERVICE)—Must specify descriptor or name

The request may return the following errors, though not as a result of standard Pascal programming practice:

ES$IAD	(type: SYSTEM_SERVICE)—Invalid address; pointer to buffer or structure is odd or not in user address space
ES$IPM	(type: SYSTEM_SERVICE)—Illegal parameter; amount of data to be sent by value (primitive's vlen parameter) exceeds packet capacity or amount of data sent by reference (primitive's rlen parameter) exceeds 8128 bytes

Applications

The COND_SEND function permits the sending process to be selective about message transmission. For example, assuming the existence of several equivalent service queues, a service-request message can be sent to the queue that has an idle server process waiting for a request.

14.10 COND_SEND_ACK

MACRO equivalent: SNDC$

The COND_SEND_ACK function implements a conditional form of the SEND_ACK request. If the function finds a process waiting (blocked) on the specified queue semaphore, this function performs the following operations:

- Allocates a packet in system space
- Copies a message buffer reference and a reply semaphore structure identifier into the packet
- Signals the queue semaphore in the same way as the PUT_PACKET request
- Returns control to the caller, with a Boolean TRUE value

The reply semaphore allows the receiver process to signal an acknowledgment to the sender.

If no process is waiting on the semaphore, the function returns control immediately to the caller, with a Boolean FALSE value.

A packet constructed by COND_SEND_ACK has the same format as one constructed by the SEND_ACK request.

The message-transmission characteristics and packet structure used by COND_SEND_ACK are identical to those provided by SEND_ACK (that is, the sending of a message reference along with a reply semaphore). The only functional difference between the two requests is the unconditional signal performed by SEND_ACK versus the conditional signal performed by COND_SEND_ACK.

Note

The message-by-reference feature must be used with caution concerning the length of the message. Once the message-copying operation begins, no other process can gain control until the entire message is copied, because of the indivisible nature of kernel primitive operations. Thus, transmission of long messages can seriously affect the servicing of interrupts by increasing interrupt latency throughout the system.

COND_SEND_ACK is intended for use by processes with general or DEV_ACCESS mapping, which cannot access a packet directly in a mapped-memory environment (see Section 10.1.2). This function permits a process, regardless of mapping type, to transmit data by reference and a reply semaphore structure identifier to another process through a packet. The RECEIVE_ACK and COND_RECEIVE_ACK requests allow any process to receive the reference data and reply semaphore transmitted by COND_SEND_ACK.

The SEND_ACK request is the unconditional form of the COND_SEND_ACK request.

Syntax

```
COND_SEND_ACK ( [[ REF_DATA := reference-data-id
                REF_LENGTH := reference-data-length ]]
                REPLY_DESC := reply-sem-descriptor
                [[ PRIORITY := packet-priority ]]
                { DESC := queue-sem-descriptor  }
                { NAME := queue-sem-name        }
                [[ STATUS := status-record ]] )
```

reference-data-id

The identifier of the variable (buffer) that contains the data to be sent by reference. The address of this variable is converted to a physical address and is placed in the packet with the reference-data-length parameter value. This parameter is significant only if the reference-data-length parameter is nonzero.

reference-data-length

A constant or the identifier of a variable of predefined type REF_DATA_LEN that specifies the length, in bytes, of the message buffer (reference-data-id parameter) containing the data to be sent by reference. The maximum value is 8128 bytes.

reply-sem-descriptor

The identifier of a variable of predefined type SEMAPHORE_DESC that contains the reply semaphore's structure identifier. The variable must have been previously initialized by a CREATE_BINARY_SEMAPHORE or a CREATE_COUNTING_SEMAPHORE, as appropriate, or by an INIT_STRUCTURE_DESC request.

packet-priority

A constant or the identifier of a variable of predefined type PRIORITY_RANGE that specifies the priority value (0 to 255) to be assigned to the packet. This value affects the order in which the packet is queued on a semaphore having a priority-ordered packet queue (see the CREATE_QUEUE_SEMAPHORE request). The default priority value is 1.

queue-sem-descriptor

The identifier of a variable of predefined type QUEUE_SEMAPHORE_DESC that contains the semaphore's structure identifier. The variable must have been previously initialized by a CREATE_QUEUE_SEMAPHORE or an INIT_STRUCTURE_DESC request.

queue-sem-name

A character-string constant or the identifier of a variable of predefined type NAME_STR that specifies the 6-character name of an existing semaphore (see Section 11.1.1.1).

status-record

The identifier of a variable of predefined record type EXC_STATUS that may receive an exception type and code. If you specify this parameter, the exception status, either success or error, that results from issuing the request is reported in this variable. Otherwise, an error causes the corresponding exception to be reported. The format of the exception record is described in Section 11.1.2.

Examples

```
VAR
  Queue_1 : QUEUE_SEMAPHORE_DESC;
  Buffer : PACKED ARRAY [0..511] OF CHAR;
  Reply : SEMAPHORE_DESC;

[PRIORITY(10), STACK_SIZE(100), NAME ('PRODUC')] PROCESS Producer;
BEGIN
```

```
  (* Conditionally send data with acknowledgment. *)
  IF COND_SEND_ACK (REF_DATA := Buffer,
                    REF_LENGTH := 512,
                    REPLY_DESC := Reply,
                    DESC := Queue_1)
   THEN WAIT (DESC := Reply);

END; (* Process Producer *)
```

Restriction

The maximum value for the reference-data-length parameter is 8128.

Semantics

The COND_SEND_ACK function tests the specified queue semaphore for a waiting process. If a process is waiting, the function performs the following actions prior to signaling the semaphore:

1. Obtains a packet from the system's free-element pool and writes the specified priority value into the packet header.
2. Constructs a control byte that consists of the reference data flag bit (r) and a 7-bit value-data-length field containing a 12; the length of the structure identifier used for the reply semaphore.
3. Places the control byte in the packet header for subsequent use by the RECEIVE_ACK and COND_RECEIVE_ACK requests and copies the reply semaphore structure identifier into the packet's value data area.
4. Constructs a physical address from the address of the message to be sent by reference, if any. This physical address is placed in the packet with the message length. A physical address consists of two words. The value of the first word is the virtual address; the value of the second word is the content of the user-mode page address register (PAR) associated with that virtual address.

If at least one process is waiting, it unblocks the first waiting process, associates the packet with that process (as its wait-return value) and calls the scheduler, which may cause the calling process to be preempted: to lose control of the CPU. On eventual return to the caller, the function returns a Boolean TRUE value to indicate a successful operation.

If no process is waiting on the semaphore, the function returns immediately to the caller, with a Boolean FALSE to indicate that the send operation was not performed.

A packet constructed by COND_SEND_ACK has the same format as one constructed by SEND_ACK, as described in Figure 14–3.

This request is implemented through the SNDC$ kernel primitive.

Error Returns

See Section 11.2 for general information about error returns. The following exception codes may be returned:

ES$CDN	(type: SYSTEM_SERVICE)—Cannot specify both descriptor and name
ES$IST	(type: SYSTEM_SERVICE)—Illegal structure descriptor; reply semaphore or queue semaphore does not exist
ES$MDN	(type: SYSTEM_SERVICE)—Must specify descriptor or name

The request may return the following errors, though not as a result of standard Pascal programming practice:

ES$IAD	(type: SYSTEM_SERVICE)—Invalid address; pointer to buffer or structure is odd or not in user address space
ES$IPM	(type: SYSTEM_SERVICE)—Illegal parameter; amount of data to be sent by value (primitive's vlen parameter) exceeds packet capacity or amount of data sent by reference (primitive's rlen parameter) exceeds 8128 bytes

14.11 CREATE_QUEUE_SEMAPHORE

MACRO equivalent: CRST$

The CREATE_QUEUE_SEMAPHORE function creates a queue semaphore structure in the system's common-memory area managed by the kernel.

If the semaphore is successfully created, the request returns a Boolean TRUE value. If not enough system memory is free to create the semaphore or if an exception occurs, the function returns a Boolean FALSE value.

The function permits a process to create a queue semaphore that can be manipulated by the various queue semaphore management requests described in this chapter.

Syntax

```
CREATE_QUEUE_SEMAPHORE ( ⟦ PROCESS_ORDER := { FIFO } ⟧
                                              { PRIO }
                         ⟦ PACKET_ORDER := { FIFO } ⟧
                                             { PRIO }
                         { DESC := queue-sem-descriptor }
                         { NAME := queue-sem-name       }
                        ⟦ STATUS := status-record ⟧ )
```

PROCESS_ORDER
The order in which waiting processes are queued in the semaphore's waiting process list. FIFO specifies first-in-first-out order and is the default value; PRIO specifies ordering by process priority.

PACKET_ORDER
The order in which the packets are kept in the semaphore's available-packet list. FIFO specifies first-in-first-out order and is the default value; PRIO specifies ordering by process priority.

queue-sem-descriptor
The identifier of a variable of predefined type QUEUE_SEMAPHORE_DESC that is to receive the semaphore's structure identifier.

queue-sem-name
A character-string constant or the identifier of a variable of predefined type NAME_STR that specifies the 6-character name of the semaphore (see Section 11.1.1.1). The name must not be the identifier of an existing process or structure.

status-record
The identifier of a variable of predefined record type EXC_STATUS that may receive an exception type and code. If you specify this parameter, the exception status, either success or error, that results from issuing the request is reported in this variable. Otherwise, an error causes the corresponding exception to be reported. The format of the exception record is described in Section 11.1.2.

If you do not specify the queue-sem-name parameter, the function creates an unnamed queue semaphore identified by the information returned in the queue-sem-descriptor variable.

Example

```
%INCLUDE 'EXC.PAS'

VAR
  Queue_1, Queue_2, Queue_3 : QUEUE_SEMAPHORE_DESC;

[INITIALIZE] PROCEDURE Init;
(* Create the needed queue semaphores.  If any create fails, then
   report an exception. *)
BEGIN

  (* Create an unnamed queue semaphore with FIFO process ordering
     and FIFO packet ordering. *)
  IF NOT CREATE_QUEUE_SEMAPHORE
          (DESC := Queue_1)
   THEN REPORT (EXC_TYPE := [RESOURCE], EXC_CODE := ES$NMK);

  (* Create  a named  queue semaphore with  FIFO process ordering
     and FIFO packet ordering. *)
  IF NOT CREATE_QUEUE_SEMAPHORE
          (DESC := Queue_2,
           NAME := 'QUEUE2')
   THEN REPORT (EXC_TYPE := [RESOURCE], EXC_CODE := ES$NMK);

  (* Create an  unnamed  queue  semaphore  with  priority process
     ordering and priority packet ordering. *)
  IF NOT CREATE_QUEUE_SEMAPHORE
          (DESC := Queue_3,
           PROCESS_ORDER := PRIO,
           PACKET_ORDER := PRIO)
   THEN REPORT (EXC_TYPE := [RESOURCE], EXC_CODE := ES$NMK);

END; (* Procedure Init *)
```

Semantics

The CREATE_QUEUE_SEMAPHORE function requests the kernel to allocate and to initialize a queue semaphore structure in the system's common memory.

If the semaphore is successfully created, the function returns a Boolean TRUE value. The semaphore is named as specified in the queue-sem-name parameter, and its structure identifier is copied into the structure descriptor record specified in the queue-sem-descriptor parameter.

If not enough system memory is free to create the semaphore, the function returns a Boolean FALSE value.

This request is implemented through the CRST$ kernel primitive.

Error Returns

See Section 11.2 for general information about error returns. The following exception codes may be returned:

ES$MDN	(type: SYSTEM_SERVICE)—Must specify structure descriptor or name
ES$SNI	(type: SYSTEM_SERVICE)—Structure name already in use; the specified name has already been given to another process, semaphore, ring buffer, logical name, or shared region

The request may return the following errors, though not as a result of standard Pascal programming practice:

ES$IAD	(type: SYSTEM_SERVICE)—Invalid address; pointer to descriptor is odd or not in user address space
ES$IPM	(type: SYSTEM_SERVICE)—Illegal parameter; an invalid structure type or argument value was specified
ES$IPR	(type: SYSTEM_SERVICE)—Illegal primitive; user code attempted to create a PCB (type ST.PCB specified)

14.12 CREATE_QUEUE_SEMAPHORE_P

MACRO equivalent: none

CREATE_QUEUE_SEMAPHORE_P creates (by a procedure) a queue semaphore structure in the system's common-memory area managed by the kernel.

If the semaphore is successfully created, the STATUS parameter is set to ES$NOR. If not enough system memory is free to create the semaphore or if an exception occurs, the STATUS parameter is set to the appropriate exception code.

The procedure permits a process to create a queue semaphore that can be manipulated by the various queue semaphore management requests described in this chapter.

Syntax

```
CREATE_QUEUE_SEMAPHORE_P ( ⟦ PROCESS_ORDER := { FIFO } ⟧
                                                { PRIO }
                           ⟦ PACKET_ORDER := { FIFO } ⟧
                                               { PRIO }
                           { DESC := queue-sem-descriptor }
                           { NAME := queue-sem-name       }
                           ⟦ STATUS := status-record ⟧ )
```

PROCESS_ORDER

The order in which waiting processes are queued in the semaphore's waiting process list. FIFO specifies first-in-first-out order and is the default value; PRIO specifies ordering by process priority.

PACKET_ORDER

The order in which the packets are kept in the semaphore's available-packet list. FIFO specifies first-in-first-out order and is the default value; PRIO specifies ordering by process priority.

queue-sem-descriptor

The identifier of a variable of predefined type QUEUE_SEMAPHORE_DESC that is to receive the semaphore's structure identifier.

queue-sem-name

A character-string constant or the identifier of a variable of predefined type NAME_STR that specifies the 6-character name of the semaphore (see Section 11.1.1.1). The name must not be the identifier of an existing process or structure.

status-record

The identifier of a variable of predefined record type EXC_STATUS that may receive an exception type and code. If you specify this parameter, the exception status, either success or error, that results from issuing the request is reported in this variable. Otherwise, an error causes the corresponding exception to be reported. The format of the exception record is described in Section 11.1.2.

If you do not specify the queue-sem-name parameter, the procedure creates an unnamed queue semaphore identified by the information returned in the queue-sem-descriptor variable.

Example

```
%INCLUDE 'EXC.PAS'
%INCLUDE 'CRPROC.PAS'

VAR
   QDESC1, QDESC2 : QUEUE_SEMAPHORE_DESC;
   P_STATUS : EXC_STATUS;
   SUCCESS : Boolean;

(* Create the queue semaphores. If any create fails, then set SUCCESS to false. *)
[INITIALIZE] PROCEDURE Init;
BEGIN
   SUCCESS := True;
   CREATE_QUEUE_SEMAPHORE_P (PROCESS_ORDER := FIFO, PACKET_ORDER := FIFO,
                            . DESC := QDESC1, NAME := 'QUEUE1',
                             STATUS := P_STATUS);
   IF (P_STATUS.EXC_CODE <> ES$NOR)
                             THEN SUCCESS := False;
   CREATE_QUEUE_SEMAPHORE_P (DESC := QDESC2, STATUS := P_STATUS);
   IF (P_STATUS.EXC_CODE <> ES$NOR)
                             THEN SUCCESS := False;
END;

BEGIN (* Main *)
   IF NOT SUCCESS
      THEN WRITELN('%ERROR - Semaphore creation failed')
   ELSE
      .
      .
      .
END.
```

Semantics

The CREATE_QUEUE_SEMAPHORE_P procedure requests the kernel to allocate and to initialize a queue semaphore structure in the system's common memory.

If the semaphore is successfully created, the STATUS parameter is set to ES$NOR. The semaphore is named as specified in the queue-sem-name parameter, and its structure identifier is copied into the structure descriptor record specified in the queue-sem-descriptor parameter.

If not enough system memory is free to create the semaphore, the STATUS parameter is set to the appropriate exception code.

This request is implemented through the CRST$ kernel primitive.

Error Returns

See Section 11.2 for general information about error returns. The following exception codes may be returned:

ES$MDN	(type: SYSTEM_SERVICE)—Must specify structure descriptor or name
ES$SNI	(type: SYSTEM_SERVICE)—Structure name already in use; the specified name has already been given to another process, semaphore, ring buffer, logical name, or shared region

The request may return the following errors, though not as a result of standard Pascal programming practice:

ES$IAD	(type: SYSTEM_SERVICE)—Invalid address; pointer to descriptor is odd or not in user address space
ES$IPM	(type: SYSTEM_SERVICE)—Illegal parameter; an invalid structure type or argument value was specified
ES$IPR	(type: SYSTEM_SERVICE)—Illegal primitive; user code attempted to create a PCB (type ST.PCB specified)

14.13 DEALLOCATE_PACKET

MACRO equivalent: DAPK$

The DEALLOCATE_PACKET procedure returns a message packet to the kernel's free-packet pool. This procedure permits the caller to release a packet acquired by means of a GET_PACKET, GET_PACKET_ANY, or a COND_GET_PACKET request when the packet is no longer needed. This procedure is for use by processes with the PRIVILEGED or DRIVER attributes or processes that reside in an unmapped-memory environment (see Sections 10.1.2 and 14.1 for more information).

The ALLOCATE_PACKET and COND_ALLOCATE_PACKET procedures permit a process to allocate (that is, obtain a pointer to) a free packet for use with PUT_PACKET and COND_PUT_PACKET requests.

Syntax

DEALLOCATE_PACKET (PACKET_PTR := pointer)

pointer

The identifier of a variable of predefined type QUEUE_PTR that contains a pointer to the packet being deallocated. The pointer must be to a word address in the kernel's data space.

Example

```
VAR
  Pack : QUEUE_PTR;
  Access : SEMAPHORE_DESC;

[PRIORITY(10), STACK_SIZE(100), NAME ('P1    ')] PROCESS P1;
BEGIN

  (* Deallocate a packet.  Also indicate we are returning one. *)
  DEALLOCATE_PACKET (PACKET_PTR := Pack);
  SIGNAL (DESC := Access);

END; (* Process P1 *)
```

Semantics

The DEALLOCATE_PACKET procedure performs the following operations:

1. If address checking is enabled in the SYSTEM macro of the kernel configuration file, the pointer parameter value is checked to ensure that it lies within the kernel's packet space. If the address is invalid, it raises an exception and returns to the calling process. If the address is valid or address checking is not enabled, the operation proceeds.
2. Returns the assumed packet to the kernel's free-packet pool.
3. If no other process is waiting for packet allocation, the procedure returns control to the calling process.
4. If at least one process is waiting for packet allocation, the newly freed packet is allocated to the highest-priority waiting process, that process is unblocked, and the scheduler is called, which may cause the calling process to be preempted, depending on the priority of the unblocked process. See the ALLOCATE_PACKET procedure.

This request is implemented through the DAPK$ kernel primitive.

Error Returns

See Section 11.2 for general information about error returns. The following exception code may be returned:

ES$IAD	(type: SYSTEM_SERVICE)—Invalid address; the pointer value is not a word address within the range of valid packet addresses

14.14 DESTROY

MACRO equivalent: DLST$

The DESTROY procedure deletes a specified structure (in this case, a queue semaphore) from the system and deallocates the memory space associated with the semaphore. The operation is performed only if no processes are blocked on the queue semaphore at the time of the call.

Syntax

```
DESTROY ( { DESC := descriptor }
          { NAME := name       }
          [[ STATUS := status-record ]] )
```

descriptor

The identifier of a variable of predefined type QUEUE_SEMAPHORE_DESC that contains the queue semaphore's structure identifier. The variable must have been previously initialized by a CREATE_QUEUE_SEMAPHORE or an INIT_STRUCTURE_DESC request.

name

A character-string constant or the identifier of a variable of predefined type NAME_STR that specifies the 6-character name of an existing semaphore.

status-record

The identifier of a variable of predefined record type EXC_STATUS that may receive an exception type and code. If you specify this parameter, the exception status, either success or error, that results from issuing the request is reported in this variable. Otherwise, an error causes the corresponding exception to be reported. The format of the exception record is described in Section 11.1.2.

Example

```
VAR
  Queue_1 : QUEUE_SEMAPHORE_DESC;

[TERMINATE] PROCEDURE Term;
BEGIN

  (* Destroy an unnamed queue semaphore. *)
  DESTROY (DESC := Queue_1);

  (* Destroy a named queue semaphore. *)
  DESTROY (NAME := 'QUEUE1  ');

END; (* Procedure Term *)
```

Semantics

If the queue semaphore is not in use, DESTROY removes its name, if one exists, from the system name table, returns the space the semaphore occupies to the free-memory pool, and returns control to the caller.

This request is implemented through the DLST$ kernel primitive.

Error Returns

See Section 11.2 for general information about error returns. The following exception codes may be returned:

ES$CDN	(type: SYSTEM_SERVICE)—Cannot specify both descriptor and name
ES$IST	(type: SYSTEM_SERVICE)—Illegal structure descriptor; no such queue semaphore exists
ES$MDN	(type: SYSTEM_SERVICE)—Must specify descriptor or name
ES$SIU	(type: SYSTEM_SERVICE)—Structure is in use and cannot be deleted

14.15 GET_PACKET

MACRO equivalent: WAIQ$

The GET_PACKET procedure waits on a specified queue semaphore for a packet. When a packet becomes available, GET_PACKET removes the packet from the semaphore's queue and returns its pointer to the caller. If no packet is available, the calling process is blocked on the semaphore, awaiting a subsequent PUT_PACKET operation. GET_PACKET permits the calling process to receive a signal from another process that a data packet on which the process is dependent is available, regardless of the order in which the PUT_PACKET and GET_PACKET requests occur. (Compare with the conditional wait-on-queue request, COND_GET_PACKET.)

This procedure is for use by processes with the PRIVILEGED or DRIVER attributes or by processes that reside in an unmapped-memory environment (see Sections 10.1.2 and 14.1 for more information).

The PUT_PACKET and COND_PUT_PACKET requests send a packet to a queue semaphore.

Syntax

```
GET_PACKET ( PACKET_PTR := pointer
             { DESC := queue-sem-descriptor }
             { NAME := queue-sem-name       }
             [[ STATUS := status-record ]] )
```

pointer

The identifier of a variable of predefined type QUEUE_PTR that will receive a pointer to the packet being obtained.

queue-sem-descriptor

The identifier of a variable of predefined type QUEUE_SEMAPHORE_DESC that contains the semaphore's structure identifier. The variable must have been previously initialized by a CREATE_QUEUE_SEMAPHORE or an INIT_STRUCTURE_DESC request.

queue-sem-name

A character-string constant or the identifier of a variable of predefined type NAME_STR that specifies the 6-character name of an existing semaphore (see Section 11.1.1.1).

status-record

The identifier of a variable of predefined record type EXC_STATUS that may receive an exception type and code. If you specify this parameter, the exception status, either success or error, that results from issuing the request is reported in this variable. Otherwise, an error causes the corresponding exception to be reported. The format of the exception record is described in Section 11.1.2.

Example

```
VAR
  Queue_1 : QUEUE_SEMAPHORE_DESC;
  Pack : QUEUE_PTR;

[PRIORITY(10), STACK_SIZE(100), NAME ('CONSUM')] PROCESS Consumer;
BEGIN

  (* Get a packet via an unnamed queue semaphore. *)
  GET_PACKET (PACKET_PTR := Pack,
              DESC := Queue_1);

  (* Get a packet via a named queue semaphore. *)
  GET_PACKET (PACKET_PTR := Pack,
              NAME := 'QUEUE1');

END; (* Process Consumer *)
```

Semantics

The GET_PACKET procedure tests for an available packet. If at least one packet is on the semaphore's queue, the procedure does the following:

1. Decrements the specified queue semaphore.
2. Removes the first available packet from the queue.
3. Returns the pointer to that packet in the variable specified by the pointer parameter.

If no packets are on the semaphore's queue, the request blocks the calling process and calls the scheduler. The calling process remains blocked until it can be reactivated by a subsequent signal of the semaphore, which places a packet on the queue.

This request is implemented through the WAIQ$ kernel primitive.

Error Returns

See Section 11.2 for general information about error returns. The following exception codes may be returned:

ES$CDN	(type: SYSTEM_SERVICE)—Cannot specify both descriptor and name
ES$IST	(type: SYSTEM_SERVICE)—Illegal structure descriptor; no such queue semaphore exists
ES$MDN	(type: SYSTEM_SERVICE)—Must specify descriptor or name

Applications

See the description of the PUT_PACKET request.

14.16 GET_PACKET_ANY

MACRO equivalent: WAQA$

The GET_PACKET_ANY function implements a complex form of the GET_PACKET request. See the GET_PACKET and PUT_PACKET requests for a description of the basic wait and signal operations on queue semaphores. GET_PACKET_ANY performs the basic wait, or "get packet," operation on the logical OR of several queue semaphores, with an optional timeout feature. That is, GET_PACKET_ANY permits the calling process to test for and, if necessary, wait on an available packet on any one of a set of queue semaphores. Up to four queue semaphores can be specified in the request.

The function returns ordinal values from 0 to 5 to indicate the results of the operation (see Semantics).

If no packet is available on any of the specified queue semaphores, the calling process blocks until any one of those semaphores is signaled and can provide a packet pointer for the calling process. (The caller could be blocked behind other waiting processes on a given queue semaphore, of course, although a multiple-waiter policy is unlikely, particularly in the case of GET_PACKET_ANY usage.)

Optionally, a GET_PACKET_ANY operation can be terminated due to the expiration of a time interval specified in the request.

Thus, GET_PACKET_ANY allows a process to get a packet pointer from any of up to four queue semaphores, each signaled by a different process, perhaps. The function might also be used primarily for its timeout capability.

If a zero time period (immediate timeout) is specified in the request, GET_PACKET_ANY provides a complex form of the COND_GET_PACKET operation, which tests for an available packet but will not block the caller. See COND_GET_PACKET for a description of the simple conditional-get-packet operation.

This function is for use by processes with the PRIVILEGED or DRIVER attributes or processes in an unmapped-memory environment (see Sections 10.1.2 and 14.1 for more information).

Syntax

```
GET_PACKET_ANY ( [[ SDB4 := queue-sem-descriptor-4 ]]
                 [[ SDB3 := queue-sem-descriptor-3 ]]
                 [[ SDB2 := queue-sem-descriptor-2 ]]
                 SDB1 := queue-sem-descriptor-1
                 PACKET_PTR := pointer
                 [[ TIMEOUT := timeout-interval ]]
                 [[ STATUS := status-record ]] )
```

queue-sem-descriptor-4
queue-sem-descriptor-3
queue-sem-descriptor-2
queue-sem-descriptor-1

The identifier of a variable of predefined type QUEUE_SEMAPHORE_DESC that contains a semaphore's structure identifier. You can specify up to four queue semaphores. The order in which you specify multiple queue semaphores determines the order in which they are initially tested for a signal. (That order can be critical under certain real-time conditions, as discussed under Semantics and Implementation Notes.)

Each descriptor must have been previously initialized by an INIT_STRUCTURE_DESC or a CREATE_QUEUE_SEMAPHORE request.

pointer

The identifier of a variable of predefined type QUEUE_PTR that will receive a pointer to the packet being obtained.

timeout-interval

The identifier of a variable of predefined type LONG_INTEGER that specifies the maximum time, in milliseconds, that the caller wishes to be blocked waiting for a signal. The value must be a positive integer from 0 to (2**31) −1. A value of 0 causes the request to time out immediately if no packet is available from any of the semaphores when GET_PACKET_ANY is called. That is, the calling process will never block if the specified time interval is 0. If you do not specify this parameter, the function assumes no timeout for the operation; the calling process may block indefinitely.

status-record

The identifier of a variable of predefined record type EXC_STATUS that may receive an exception type and code. If you specify this parameter, the exception status, either success or error, that results from issuing the request is reported in this variable. Otherwise, an error causes the corresponding exception to be reported. The format of the exception record is described in Section 11.1.2.

Restrictions

- The timeout-interval value is limited to a 31-bit positive integer; that is, the sign bit of the high-order word must not be set. (The maximum valid value, in milliseconds, permits a timeout period of just over 24.89 days; see the SLEEP procedure for more detail.)
- If you wish to use fewer than four queue semaphore descriptor parameters, you must assign them beginning with keyword SDB1. You may not assign keyword parameters with higher-numbered suffixes unless all keywords with lower-numbered suffixes are assigned. For example, if the parameter sequence specifies keyword SDB3, the sequence must also include keywords SDB2 and SDB1.

Example

```
%INCLUDE 'COMPLX.PAS'

VAR
  Queue_1, Queue_2, Queue_3 : QUEUE_SEMAPHORE_DESC;
  Pack : QUEUE_PTR;
  Which_one : COMPLEX_FUNC_VALUE;
  Timeout_val : LONG_INTEGER;

[PRIORITY(10), STACK_SIZE(100), NAME ('CONSUM')] PROCESS Consumer;
BEGIN
```

```
  (* Get a packet from one of three queue semaphores. *)
  Which_one := GET_PACKET_ANY
                   (PACKET_PTR := Pack,
                    SDB1 := Queue_1,
                    SDB2 := Queue_2,
                    SDB3 := Queue_3);

  (* Get a packet from one of two queue semaphores with a timeout. *)
  Timeout_val := 1000;
  Which_one := GET_PACKET_ANY
                   (PACKET_PTR := Pack,
                    SDB1 := Queue_1,
                    SDB2 := Queue_2,
                    TIMEOUT := Timeout_val);

END; (* Process Consumer *)
```

Semantics

The GET_PACKET_ANY function tests each of the queue semaphores specified in the request for an available queue element or message packet. The queue semaphores are tested in the keyword order: SDB1 to SDB4.

The function returns the following values:

Value	Meaning
0	Request timed out
1	Request satisfied by SDB1
2	Request satisfied by SDB2
3	Request satisfied by SDB3
4	Request satisfied by SDB4
5	Error condition

If any of the semaphores has a packet at the time of the call, the function dequeues a packet pointer from the first such semaphore encountered and returns the pointer immediately to the caller along with a function return value between 1 and 4 that indicates which queue semaphore specified in the call satisfied the request.

If none of the semaphores has a packet and either no timeout-interval or a nonzero timeout-interval was supplied in the call, the function switches the calling process to the wait-active state. In that state, the process is blocked on all the semaphores specified in the request.

If none of the semaphores has a packet and a zero timeout-interval was supplied in the call, the function returns immediately to the caller with a zero value, indicating a return due to timeout. Thus, in the case of an immediate timeout, the calling process never leaves the run state.

If the calling process switches to the wait-active state, the process is blocked from execution until it can be reactivated either by a packet becoming available on one of the blocking semaphores (see PUT_PACKET semantics) or by elapse of the specified timeout period, if any. When reactivated for either reason, the process is unblocked from all the semaphores and is switched to either the ready-active or run state, depending on relative process priorities. If unblocked because of an available packet, the function returns the ordinal value (from 1 to 4) of the semaphore that triggered the return, as described above. If unblocked because of a timeout, the function returns a 0.

If an error occurs, the function returns a 5.

This request is implemented through the WAQA$ kernel primitive.

Error Returns

See Section 11.2 for general information about error returns. The following exception code may be returned:

ES$IST	(type: SYSTEM_SERVICE)—Invalid structure descriptor; no such queue semaphore exists

The request may return the following error, though not as a result of standard Pascal programming practice:

ES$IPM	(type: SYSTEM_SERVICE)—Illegal parameter; timeout value out of range

Implementation Notes

Since the initial test of the semaphores for an available packet is performed in determinate order, the order in which multiple semaphores are identified in the call can be critical under certain real-time conditions. For example, assume that the relative frequency of signals or sends is high for one of several queue semaphores and that the "fast" queue semaphore is identified as the first to be tested for signals by being associated with keyword SDB1.

In a series of calls to GET_PACKET_ANY, that semaphore will be serviced far more often than the others, and the "slower" semaphores may seldom or never be tested and serviced. Optimally, then, the semaphore with the highest expected signal rate should be assigned to the keyword that is tested last; the next highest as next to last, and so on, assuming that probable relative frequencies can be determined. Alternatively, the order in which the semaphores are identified could be rotated in successive calls so at least N semaphores are guaranteed to be tested in N calls to GET_PACKET_ANY. The correct or best-case strategy depends on application-specific factors, of course.

14.17 GET_VALUE

MACRO equivalent: GVAL$

The GET_VALUE procedure obtains the value and type code of a specified structure. The code identifies a structure as a binary, counting, or queue semaphore or as a ring buffer. The meaning of the structure's value depends on the structure type. For example, the value of a counting semaphore is the current signal count, whereas the value of a ring buffer is the current element count.

Note

The value of a structure may change immediately after it is inspected. Therefore, the information this request provides must be used cautiously, to prevent the introduction of race conditions.

Syntax

```
GET_VALUE (  VALUE := count
             TYP := structure-type
             { DESC := descriptor }
             { NAME := name       }
             [[ STATUS := status-record ]] )
```

count

The identifier of a variable of type INTEGER that receives the structure's value.

structure-type

The identifier of a variable of type INTEGER that receives the structure's type code.

descriptor

The identifier of a variable of predefined type QUEUE_SEMAPHORE_DESC that contains the semaphore's structure identifier. The variable must have been previously initialized by an appropriate CREATE-type request or an INIT_STRUCTURE_DESC request.

name

A character-string constant or the identifier of a variable of predefined type NAME_STR that specifies the 6-character name of an existing semaphore.

status-record

The identifier of a variable of predefined record type EXC_STATUS that may receive an exception type and code. If you specify this parameter, the exception status, either success or error, that results from issuing the request is reported in this variable. Otherwise, an error causes the corresponding exception to be reported. The format of the exception record is described in Section 11.1.2.

Structure Type Identification Codes

The type codes and meaning of the values that the procedure can return are:

Structure	Type Code	Meaning of Value Parameter
Binary semaphore	0	The value of the gate variable (0 or 1)
Counting semaphore	1	The count of pending signals (0 or positive)
Queue semaphore	2	The count of pending signals (0 or positive)
Ring buffer	3	The count of data elements in the ring buffer
PCB	4	No meaning
SRD	5	No meaning
Unformatted	7	No meaning

Example

```
VAR
  Sem_val, Sem_typ : INTEGER;
  Queue_1 : QUEUE_SEMAPHORE_DESC;

[PRIORITY(10), STACK_SIZE(100)] PROCESS P1;
BEGIN

  (* Get the value of an unnamed queue semaphore. *)
  GET_VALUE (VALUE := Sem_val, TYP := Sem_typ, DESC := Queue_1);

  (* Get the value of a named queue semaphore. *)
  GET_VALUE (VALUE := Sem_val, TYP := Sem_typ, NAME := 'QUEUE1   ');

END; (* Process P1 *)
```

Semantics

The GET_VALUE procedure obtains the type code and value of the specified structure, stores that information in the variables specified in the call, and returns control to the caller.

This request is implemented through the GVAL$ kernel primitive.

Error Returns

See Section 11.2 for general information about error returns. The following exception codes may be returned:

ES$CDN	(type: SYSTEM_SERVICE)—Cannot specify both descriptor and name
ES$IPR	(type: SYSTEM_SERVICE)—Illegal primitive; the descriptor or name parameter is a logical name that does not translate to the name of a structure
ES$IST	(type: SYSTEM_SERVICE)—Illegal structure descriptor; specified structure does not exist
ES$MDN	(type: SYSTEM_SERVICE)—Must specify descriptor or name

14.18 INIT_STRUCTURE_DESC

MACRO equivalent: GVAL$

The INIT_STRUCTURE_DESC procedure copies identifying information about a specified queue semaphore into a structure descriptor record. This record provides the kernel with a rapid-access path to a semaphore referred to in the other queue semaphore management requests described in this chapter.

You may also set up a structure descriptor record when you create a queue semaphore, using the CREATE_QUEUE_SEMAPHORE request.

Syntax

```
INIT_STRUCTURE_DESC (  DESC := descriptor
                       NAME := name
                       [[ STATUS := status-record ]] )
```

descriptor

The identifier of a variable of predefined type QUEUE_SEMAPHORE_DESC that is to receive the semaphore's structure identifier.

name

A character-string constant or the identifier of a variable of predefined type NAME_STR that specifies the 6-character name of an existing semaphore.

status-record

The identifier of a variable of predefined record type EXC_STATUS that may receive an exception type and code. If you specify this parameter, the exception status, either success or error, that results from issuing the request is reported in this variable. Otherwise, an error causes the corresponding exception to be reported. The format of the exception record is described in Section 11.1.2.

Example

```
VAR
  Queue_1 : QUEUE_SEMAPHORE_DESC;

[PRIORITY(10), STACK_SIZE(100), NAME ('P1    ')] PROCESS P1;
BEGIN

  (* Get the id of the queue semaphore named 'QUEUE1  '. *)
  INIT_STRUCTURE_DESC (DESC := Queue_1, NAME := 'QUEUE1  ');

END; (* Process P1 *)
```

Semantics

The INIT_STRUCTURE_DESC procedure requests the kernel to copy the structure identifier, consisting of the index and serial number associated with the structure named in the name parameter, into the structure descriptor record specified in the descriptor parameter.

This request is implemented through the GVAL$ kernel primitive.

Error Returns

See Section 11.2 for general information about error returns. The following exception code may be returned:

ES$IST	(type: SYSTEM_SERVICE)—Illegal structure descriptor; no such queue semaphore exists

14.19 PUT_PACKET

MACRO equivalent: SGLQ$

The PUT_PACKET procedure places a packet on the queue of a specified queue semaphore and signals that semaphore. If any processes are waiting on that semaphore, the first one is unblocked, and a pointer to the packet is eventually passed to that process. (The packet is dequeued in this case.) If no process is waiting, the packet remains on the queue, and the signal remains in effect.

PUT_PACKET permits the calling process to signal another process that a data packet the process needs or will need is available, whether or not that process is waiting for the signal. (Compare with the conditional form of the request, COND_PUT_PACKET.)

This procedure is for use by processes with the PRIVILEGED or DRIVER attributes or by processes that reside in an unmapped-memory environment (see Sections 10.1.2 and 14.1 for more information).

The GET_PACKET and COND_GET_PACKET requests allow a process to obtain a packet from a queue semaphore.

Syntax

```
PUT_PACKET (  PACKET_PTR := pointer
             { DESC := queue-sem-descriptor }
             { NAME := queue-sem-name       }
             [[ STATUS := status-record ]] )
```

pointer

The identifier of a variable of predefined type QUEUE_PTR that contains a pointer to the packet being sent.

queue-sem-descriptor

The identifier of a variable of predefined type QUEUE_SEMAPHORE_DESC that contains the semaphore structure identifier. The variable must have been previously initialized by a CREATE_QUEUE_SEMAPHORE or an INIT_STRUCTURE_DESC request.

queue-sem-name

A character-string constant or the identifier of a variable of predefined type NAME_STR that specifies the 6-character name of an existing semaphore (see Section 11.1.1.1).

status-record

The identifier of a variable of predefined record type EXC_STATUS that may receive an exception type and code. If you specify this parameter, the exception status, either success or error, that results from issuing the request is reported in this variable. Otherwise, an error causes the corresponding exception to be reported. The format of the exception record is described in Section 11.1.2.

Example

```
VAR
  Queue_1 : QUEUE_SEMAPHORE_DESC;
  Pack : QUEUE_PTR;

[PRIORITY(10), STACK_SIZE(100), NAME ('PRODUC')] PROCESS Producer;
BEGIN

  (* Send a packet via an unnamed queue semaphore. *)
  PUT_PACKET (PACKET_PTR := Pack,
              DESC := Queue_1);

  (* Send a packet via a named queue semaphore. *)
  PUT_PACKET (PACKET_PTR := Pack,
              NAME := 'QUEUE1  ');

END; (* Process Producer *)
```

Semantics

The PUT_PACKET request tests the specified queue semaphore for waiting processes. If no process is waiting, the request signals the semaphore, links the passed packet into the queue, and returns to the caller.

If at least one process is waiting, the request unblocks the first waiting process, associates the packet with that process, and calls the scheduler, which may cause the calling process to be preempted: to lose control of the CPU.

This request is implemented through the SGLQ$ kernel primitive.

Error Returns

See Section 11.2 for general information about error returns. The following exception codes may be returned:

ES$CDN	(type: SYSTEM_SERVICE)—Cannot specify both descriptor and name
ES$IST	(type: SYSTEM_SERVICE)—Illegal structure descriptor; no such queue semaphore exists
ES$MDN	(type: SYSTEM_SERVICE)—Must specify descriptor or name

Applications

Queue semaphores may be used to implement general queueing functions. The PUT_PACKET request places a packet on a queue, where the packet remains until another process removes the packet with either a GET_PACKET or a receive operation. This basic mechanism can be used to implement a simple message facility or a generalized queued I/O facility.

14.20 RECEIVE

MACRO equivalent: RCVD$

The RECEIVE procedure waits on a specified queue semaphore until a packet becomes available, then copies the data in the packet into the caller's buffer space. After the data is copied, the packet is returned to the system's free-element pool for reuse. The packet format expected by RECEIVE is the same as that produced by the SEND and COND_SEND requests, as described in Figure 14–2.

A message sent by value (up to 34 bytes in length) is copied by the RECEIVE request from the packet to the receiver's buffer. In the case of a message sent by reference (possibly longer than 34 bytes), the message is ordinarily copied from the sender's message buffer, which is described in the packet, to a buffer specified by the receiver. If no message-by-reference buffer is specified in the receive request, the message is not copied, but the request returns the message reference to the caller.

Note

The message-by-reference feature must be used with caution concerning the length of the message. Once the message-copying operation begins, no other process can gain control until the entire message is copied, because of the indivisible nature of kernel primitive operations. Thus, transmission of long messages can seriously affect the servicing of interrupts by increasing interrupt latency throughout the system.

RECEIVE is intended for use by processes with general or DEV_ACCESS mapping, which cannot access a packet directly in a mapped-memory environment (see Section 10.1.2). This function permits a process, regardless of mapping type, to obtain data from another process through a packet.

The COND_RECEIVE request is the conditional, or nonblocking, form of the RECEIVE request.

The SEND and COND_SEND requests permit any type of process to transmit data through a packet.

Syntax

```
RECEIVE ( [[ REF_DATA := reference-data-id
             REF_LENGTH := reference-data-length ]]
          [[ VAL_DATA := value-data-id
             VAL_LENGTH := value-data-length ]]
          [[ RET_INFO := information-record ]]
          { DESC := queue-sem-descriptor }
          { NAME := queue-sem-name       }
          [[ STATUS := status-record ]] )
```

reference-data-id

The identifier of the variable (buffer) that will contain the data being received by reference. This parameter is significant only if the reference-data-length parameter is nonzero.

reference-data-length

A constant or the identifier of a variable of type REF_DATA_LEN that specifies the length, in bytes, of the message buffer (reference-data-id parameter) containing the data to be received by reference. This parameter limits the amount of data to be copied from the sender's buffer. If the value is 0 and if a message by reference exists in the packet, the message is not copied; the reference is passed to the receiver in the record specified by the information-record parameter. The maximum value is 8128 bytes.

value-data-id

The identifier of the variable (buffer) that will contain the data being received by value. The contents of this buffer are copied directly from the packet. This parameter is significant only if the value-data-length parameter is nonzero.

value-data-length

A constant or the identifier of a variable of type VAL_DATA_LEN that specifies the length, in bytes, of the buffer identified by the value-data-id parameter. This parameter limits the amount of data to be copied from the packet. The value of this parameter can be from 0 to 34. (See Restrictions.) The default is 0.

information-record

The identifier of a variable of predefined type INFO_BLOCK that may receive status information about the receive operation.

queue-sem-descriptor

The identifier of a variable of predefined type QUEUE_SEMAPHORE_DESC that contains the semaphore's structure identifier. The variable must have been previously initialized by a CREATE_QUEUE_SEMAPHORE or an INIT_STRUCTURE_DESC request.

queue-sem-name

A character-string constant or the identifier of a variable of predefined type NAME_STR that specifies the 6-character name of an existing semaphore (see Section 11.1.1.1).

status-record

The identifier of a variable of predefined record type EXC_STATUS that may receive an exception type and code. If you specify this parameter, the exception status, either success or error, that results from issuing the request is reported in this variable. Otherwise, an error causes the corresponding exception to be reported. The format of the exception record is described in Section 11.1.2.

Format of Information Record

The information returned to the caller in the variable specified by the information-record parameter is a record of predefined type INFO_BLOCK as follows:

```
INFO_BLOCK = PACKED RECORD
                PRIORITY       : [BYTE] PRIORITY_RANGE;
                VAL_XMIT_LEN   : [BYTE] VAL_DATA_LEN;
                ADDRESS        : PHYSICAL_ADDRESS
                REF_XMIT_LEN   : [WORD] REF_DATA_LEN;
              END;
```

PRIORITY

The priority value that was assigned to the packet by the send operation.

VAL_XMIT_LEN

The number of bytes that were sent by value. This value may be greater than the number of bytes received, which is limited by the value-data-length parameter. A 0 indicates that no data by value was sent.

ADDRESS

A record of predefined type PHYSICAL_ADDRESS that contains the physical address of the sender's message-by-reference buffer, if any. The format of the record is:

```
PHYSICAL_ADDRESS = PACKED RECORD
                      ADDRESS     : UNSIGNED;
                      PAR_VALUE   : UNSIGNED;
                   END;
```

ADDRESS

The address within the sender's address space of the message-by-reference buffer. This return value is valid only if the REF_XMIT_LEN value is nonzero; otherwise, the contents of this word are unpredictable.

PAR_VALUE

The value of the page address register (PAR) that maps the sender's message-by-reference buffer, if any. This return value is valid only in a mapped-memory environment when the REF_XMIT_LEN value is nonzero; otherwise, the contents of this word are unpredictable.

REF_XMIT_LEN

The number of bytes that were sent by reference. This value may be greater than the number of bytes received, which is limited by the reference-data-length parameter. Zero indicates that no data by reference was sent.

Restrictions

- The maximum value for the reference-data-length parameter is 8128.
- A total of 34 bytes is available in a packet for message data, a message reference, or both. A reference occupies three words in the packet and, if included, reduces the space available for data by value to 28 bytes.

Example

```
VAR
  Queue_1 : QUEUE_SEMAPHORE_DESC;
  Buffer : PACKED ARRAY [0..511] OF CHAR;
  Length : 0..512;
  Info : INFO_BLOCK;

[PRIORITY(10), STACK_SIZE(100), NAME ('CONSUM')] PROCESS Consumer;
BEGIN
```

```
  (* Receive data by reference. *)
  RECEIVE (REF_DATA := Buffer,
           REF_LENGTH := Length,
           RET_INFO := Info,
           DESC := Queue_1);

  (* Receive data by value. *)
  RECEIVE (VAL_DATA := Buffer,
           VAL_LENGTH := Length,
           RET_INFO := Info,
           DESC := Queue_1);

END; (* Process Consumer *)
```

Semantics

The RECEIVE procedure decrements the specified queue semaphore and tests for an available packet. If at least one packet is on the semaphore's queue, RECEIVE removes the first available packet from the queue and performs the following operations:

1. Copies data sent by value, if any, from the packet in system space to the caller's value-data-id buffer area. The number of bytes copied is the lesser of the value-data-length parameter value and the number of bytes sent by value.
2. Copies data sent by reference, if any, from the sender's message buffer to the caller's reference-data-id buffer area. The number of bytes copied is the lesser of the reference-data-length parameter value and REF_XMIT_LENGTH field returned in the variable specified by the information-record parameter.
3. Copies the priority of the packet and the number of bytes sent by value from the packet header to the PRIORITY and VAL_XMIT_LEN fields of the receiver's information-record area.
4. Copies the message reference, if any, contained in the packet to the corresponding three words of the receiver's information return area.
5. Zeros the REF_XMIT_LEN field of the receiver's information-record area if the packet contains no message reference.
6. Deallocates the packet, returning it to the system's free-element pool for reuse.

If no packets are on the semaphore's queue, RECEIVE blocks the calling process and calls the scheduler.

The calling process remains blocked until it can be reactivated by a subsequent signal of the semaphore that places a packet on the queue.

The packet format expected by the RECEIVE request is described in Figure 14–2.

This request is implemented through the RCVD$ kernel primitive.

Error Returns

See Section 11.2 for general information about error returns. The following exception codes may be returned:

ES$CDN	(type: SYSTEM_SERVICE)—Cannot specify both descriptor and name
ES$IST	(type: SYSTEM_SERVICE)—Illegal structure descriptor; no such queue semaphore exists
ES$MDN	(type: SYSTEM_SERVICE)—Must specify descriptor or name

The request may return the following error, though not as a result of standard Pascal programming practice:

ES$IAD	(type: SYSTEM_SERVICE)—Invalid address; pointer to buffer, rtnptr is not a word-address (even) value

14.21 RECEIVE_ACK

MACRO equivalent: RCVD$

The RECEIVE_ACK procedure waits on a specified queue semaphore until a packet becomes available. If a packet is available, RECEIVE_ACK performs the following operations:

- Copies any referenced data from the sender's buffer to the receiver's buffer
- Signals the reply semaphore automatically or passes the structure identifier of a reply semaphore to the receiver for manual signaling if the sender specified a reply semaphore

If no packet is available, the calling process blocks until a packet becomes available.

The packet format expected by RECEIVE_ACK is the same as that produced by the SEND_ACK and COND_SEND_ACK requests, as described in Figure 14–3.

Note

The message-by-reference feature must be used with caution concerning the length of the message. Once the message-copying operation begins, no other process can gain control until the entire message is copied, because of the indivisible nature of kernel primitive operations. Thus, transmission of long messages can seriously affect the servicing of interrupts by increasing interrupt latency throughout the system.

RECEIVE_ACK is intended for use by processes with general or DEV_ACCESS mapping, which cannot access a packet directly in a mapped-memory environment (see Section 10.1.2). This function permits a process, regardless of mapping type, to obtain data by reference and a reply semaphore from another process through a packet.

The SEND_ACK and COND_SEND_ACK requests permit a process to transmit data by reference and a reply semaphore through a packet.

The COND_RECEIVE_ACK request is the conditional, or nonblocking, form of the RECEIVE_ACK request.

Syntax

```
RECEIVE_ACK (  [[ REF_DATA := reference-data-id
                  REF_LENGTH := reference-data-length ]]
               [[ REPLY_DESC := reply-sem-descriptor ]]
               REC_LENGTH := ref-xmit-length
               { DESC := queue-sem-descriptor }
               { NAME := queue-sem-name       }
               [[ STATUS := status-record ]] )
```

reference-data-id

The identifier of the variable (buffer) that will contain the data being received by reference. This parameter is significant only if the reference-data-length parameter is nonzero.

reference-data-length

A constant or the identifier of a variable of predefined type REF_DATA_LEN that specifies the length, in bytes, of the message buffer (reference-data-id parameter) containing the data to be received by reference. This parameter limits the amount of data to be copied from the sender's buffer. If the value is 0 and if a message by reference exists in the packet, the message is not copied. The maximum value is 8128 bytes.

reply-sem-descriptor
The identifier of a variable of predefined type SEMAPHORE_DESC that will receive the structure identifier of the binary or counting reply semaphore provided by the sender. If you do not specify this parameter, the procedure automatically signals the reply semaphore when the receive operation is complete. Otherwise, the reply semaphore may be signaled manually at the receiver's discretion.

ref-xmit-length
The identifier of a variable of predefined type REF_DATA_LEN that will receive a value that is the number of bytes of data sent by reference. This value may be greater than the number of bytes received, which is limited by the reference-data-length parameter. Zero indicates that no data by reference was sent.

queue-sem-descriptor
The identifier of a variable of predefined type QUEUE_SEMAPHORE_DESC that contains the semaphore's structure identifier. The variable must have been previously initialized by a CREATE_QUEUE_SEMAPHORE or an INIT_STRUCTURE_DESC request.

queue-sem-name
A character-string constant or the identifier of a variable of predefined type NAME_STR that specifies the 6-character name of an existing semaphore (see Section 11.1.1.1).

status-record
The identifier of a variable of predefined record type EXC_STATUS that may receive an exception type and code. If you specify this parameter, the exception status, either success or error, that results from issuing the request is reported in this variable. Otherwise, an error causes the corresponding exception to be reported. The format of the exception record is described in Section 11.1.2.

Examples

```
VAR
  Queue_1 : QUEUE_SEMAPHORE_DESC;
  Buffer : PACKED ARRAY [0..511] OF CHAR;
  Length : 0..512;
  Received : REF_DATA_LEN;
  Reply : SEMAPHORE_DESC;

[PRIORITY(10), STACK_SIZE(100), NAME ('CONSUM')] PROCESS Consumer;
BEGIN

  (* Receive data with acknowledgment. *)
  RECEIVE_ACK (REF_DATA := Buffer,
               REF_LENGTH := Length,
               REPLY_DESC := Reply,
               REC_LENGTH := Received,
               DESC := Queue_1);

  (* Signal the reply semaphore. *)
  SIGNAL (DESC := Reply);

END; (* Process Consumer *)
```

Restrictions

The maximum value for the reference-data-length parameter is 8128.

Semantics

The RECEIVE_ACK procedure decrements the specified queue semaphore and tests for an available packet. If at least one packet is on the semaphore's queue, RECEIVE_ACK removes the first available packet from the queue and performs the following operations:

1. Copies data sent by reference, if any, from the sender's message buffer to the caller's reference-data-id buffer area. The number of bytes copied is the lesser of the reference-data-length parameter value and the ref-xmit-length parameter value.
2. Returns a 0 in the variable specified by the ref-xmit-length parameter if the message reference in the packet contained no data.
3. Signals the reply semaphore automatically if the receiver did not specify the reply-sem-descriptor parameter and if the sender passed a structure identifier in the packet. The procedure returns the reply semaphore structure identifier if the receiver did specify the reply-sem-descriptor parameter and if the sender passed a structure identifier in the packet.
4. Deallocates the packet, returning it to the system's free-element pool for reuse.

If no packets are on the semaphore's queue, the RECEIVE_ACK blocks the calling process and calls the system's scheduler.

The calling process remains blocked until it can be reactivated by a subsequent signal of the semaphore that places a packet on the queue.

The packet format expected by the RECEIVE_ACK request is described in Figure 14–3.

This request is implemented through the RCVD$ kernel primitive.

Error Returns

See Section 11.2 for general information about error returns. The following exception codes may be returned:

ES$CDN	(type: SYSTEM_SERVICE)—Cannot specify both descriptor and name
ES$IST	(type: SYSTEM_SERVICE)—Illegal structure descriptor; no such queue semaphore exists
ES$MDN	(type: SYSTEM_SERVICE)—Must specify descriptor or name
ES$RDE	(type: SYSTEM_SERVICE)—Reply descriptor expected

The request may return the following error, though not as a result of standard Pascal programming practice:

ES$IAD	(type: SYSTEM_SERVICE)—Invalid address; pointer to buffer, or rtnptr is not a word-address (even) value

14.22 RECEIVE_ANY

MACRO equivalent: RCVA$

The RECEIVE_ANY function implements a complex form of the RECEIVE operation; see the RECEIVE and SEND requests for a description of the basic receive and send message operations on queue semaphores. RECEIVE_ANY performs the basic receive operation on the logical OR of several queue semaphores, with an optional timeout feature. That is, RECEIVE_ANY permits the calling process to test for and, if necessary, wait on message data on any one of a set of queue semaphores. Up to four queue semaphores can be specified in the request.

The function returns ordinal values from 0 to 5 to indicate the results of the operation (see Semantics).

If no message packet is available on any of the specified semaphores, the calling process blocks until any one of those semaphores is signaled (or sent to) and provides a message for the calling process. (The caller could be blocked behind other waiting processes on a given queue semaphore, of course, although a multiple-receiver policy is unlikely, particularly in the case of RECEIVE_ANY usage.) The caller receives message data by value or by reference, or by a combination of both, as described for the basic RECEIVE operation.

Optionally, a RECEIVE_ANY operation can be terminated due to the expiration of a time interval specified in the request.

Thus, RECEIVE_ANY allows a process to get a message from any of up to four queue semaphores, each semaphore being signaled (put or sent to) by a separate process, for example. The function might also be used primarily for its timeout capability.

If a zero time period (immediate timeout) is specified in the request, RECEIVE_ANY provides a complex form of the COND_RECEIVE request, which tests for an available message but will not block the caller.

Note

The message-by-reference feature must be used with caution concerning the length of the message. Once the message-copying operation begins, no other process can gain control until the entire message is copied, because of the indivisible nature of kernel primitive operations. Thus, transmission of long messages can seriously affect the servicing of interrupts by increasing interrupt latency throughout the system.

RECEIVE_ANY is intended for use by processes with general or DEV_ACCESS mapping, which cannot access a packet directly in a mapped-memory environment (see Sections 10.1.2 and 14.1 for more information). The function permits a process, regardless of mapping type, to obtain data from another process through a packet.

See COND_RECEIVE for a description of the basic conditional-receive operation.

Syntax

```
RECEIVE_ANY (  [[ SDB4 := queue-sem-descriptor-4 ]]
               [[ SDB3 := queue-sem-descriptor-3 ]]
               [[ SDB2 := queue-sem-descriptor-2 ]]
               SDB1 := queue-sem-descriptor-1
               [[ REF_DATA := reference-data-id
               REF_LENGTH := reference-data-length ]]
               [[ VAL_DATA := value-data-id
               VAL_LENGTH := value-data-length ]]
               [[ RET_INFO := information-record ]]
               [[ TIMEOUT := timeout-interval ]]
               [[ STATUS := status-record ]] )
```

queue-sem-descriptor-4
queue-sem-descriptor-3
queue-sem-descriptor-2
queue-sem-descriptor-1

The identifier of a variable of predefined type QUEUE_SEM_DESC that contains a semaphore's structure identifier. You can specify up to four queue semaphores. The order in which you specify multiple queue semaphores determines the order in which they are initially tested for a signal. (That order can be critical under certain real-time conditions, as discussed under Semantics and Implementation Notes.)

Each variable must have been previously initialized by a CREATE_QUEUE_SEMAPHORE or an INIT_STRUCTURE_DESC request.

reference-data-id

The identifier of the variable (buffer) that will receive the data being sent by reference. This parameter is significant only if the reference-data-length parameter is nonzero.

reference-data-length

A constant or the identifier of a variable of type REF_DATA_LEN that specifies the length, in bytes, of the buffer identified by the reference-data-id parameter that will receive the data sent by reference. This parameter limits the amount of data to be copied from the sender's buffer. If the value is 0 and if a message by reference exists in the packet, the message is not copied; the reference is passed to the receiver in the record specified by the information-record parameter. The maximum value is 8128 bytes.

value-data-id

The identifier of the variable (buffer) that will receive the data being sent by value. The contents of this buffer are copied directly from the packet. This parameter is significant only if the value-data-length parameter is nonzero.

value-data-length

A constant or the identifier of a variable of type VAL_DATA_LEN that specifies the length, in bytes, of the buffer identified by the value-data-id parameter. This parameter limits the amount of data to be copied from the packet. The value of this parameter can be from 0 to 34. (See Restrictions.) The default value is 0.

information-record

The identifier of a variable of predefined type INFO_BLOCK that may receive status information about the receive operation.

timeout-interval

The identifier of a variable of predefined type LONG_INTEGER that specifies the maximum time, in milliseconds, that the caller wishes to be blocked waiting for data. The value must be a positive integer from 0 to (2**31) −1. A value of 0 causes the request to time out immediately if no packet is available from any of the specified semaphores when RECEIVE_ANY is called. That is, the calling process will never block if the specified time interval is 0. If you do not specify this parameter, the function assumes no timeout for the operation; the calling process may block indefinitely.

status-record

The identifier of a variable of predefined record type EXC_STATUS that may receive an exception type and code. If you specify this parameter, the exception status, either success or error, that results from issuing the request is reported in this variable. Otherwise, an error causes the corresponding exception to be reported. The format of the exception record is described in Section 11.1.2.

Format of Information Record

The information returned to the caller in the variable specified by the information-record parameter is a record of predefined type INFO_BLOCK as follows:

```
INFO_BLOCK = PACKED RECORD
                PRIORITY       : [BYTE] PRIORITY_RANGE;
                VAL_XMIT_LEN   : [BYTE] VAL_DATA_LEN;
                ADDRESS        : PHYSICAL_ADDRESS
                REF_XMIT_LEN   : [WORD] REF_DATA_LEN;
             END;
```

PRIORITY

The priority value that was assigned to the packet by the send operation.

VAL_XMIT_LEN

The number of bytes that were sent by value. This value may be greater than the number of bytes received, which is limited by the value-data-length parameter. Zero indicates that no data by value was sent.

ADDRESS

A record of predefined type PHYSICAL_ADDRESS that contains the physical address of the sender's message-by-reference buffer, if any. With suitable manipulation, this information could be used to construct a RIB for a MAP_WINDOW operation. The format of the record is:

```
PHYSICAL_ADDRESS = PACKED RECORD
                      ADDRESS     : UNSIGNED;
                      PAR_VALUE   : UNSIGNED;
                   END;
```

ADDRESS

The virtual address of the message-by-reference buffer in the sender's address space. This return value is valid only if the REF_XMIT_LEN value is nonzero; otherwise, the contents of this word are unpredictable.

PAR_VALUE

The value of the page address register (PAR) that maps the sender's message-by-reference buffer, if any. This return value is valid only in a mapped-memory environment when the REF_XMIT_LEN value is nonzero; otherwise, the contents of this word are unpredictable.

REF_XMIT_LEN

The number of bytes that were sent by reference. This value may be greater than the number of bytes received, which is limited by the reference-data-length parameter. Zero indicates that no data by reference was sent.

Restrictions

- The maximum value for the reference-data-length parameter is 8128.
- A total of 34 bytes is available in a packet for message data, a message reference, or both. A reference occupies three words in the packet and, if included, reduces the space available for data by value to 28 bytes.
- The timeout-interval value is limited to a 31-bit positive integer; that is, the sign bit of the high-order word must not be set. (The maximum valid value, in milliseconds, permits a timeout period of just over 24.89 days; see the SLEEP procedure for more detail.)
- If you wish to use fewer than four queue semaphore descriptor parameters, you must assign them beginning with keyword SDB1. You may not assign keyword parameters with higher-numbered suffixes unless all keywords with lower-numbered suffixes are assigned. For example, if the parameter sequence specifies keyword SDB3, the sequence must also include keywords SDB2 and SDB1.

Example

```
%INCLUDE 'COMPLX.PAS'

VAR
  Queue_1, Queue_2, Queue_3 : QUEUE_SEMAPHORE_DESC;
  Buffer : PACKED ARRAY [0..511] OF CHAR;
  Length : 0..512;
  Info : INFO_BLOCK;
  Which_one : COMPLEX_FUNC_VALUE;
  Timeout_val : LONG_INTEGER;

[PRIORITY(10), STACK_SIZE(100), NAME ('CONSUM')] PROCESS Consumer;
BEGIN

  (* Receive data by reference from one of three queue semaphores. *)
  Which_one := RECEIVE_ANY
                 (REF_DATA := Buffer,
                  REF_LENGTH := Length,
                  RET_INFO := Info,
                  SDB1 := Queue_1,
                  SDB2 := Queue_2,
                  SDB3 := Queue_3);
  (* Receive data by value from one of two queue semaphores *)
  (* with a timeout. *)
  Timeout_val := 1000;
  Which_one := RECEIVE_ANY
                (VAL_DATA := Buffer,
                 VAL_LENGTH := Length,
                 RET_INFO := Info,
                 SDB1 := Queue_1,
                 SDB2 := Queue_2,
                 TIMEOUT := Timeout_val);

END; (* Process Consumer *)
```

Semantics

The RECEIVE_ANY function tests each of the queue semaphores specified in the request for an available queue element, or message packet. The queue semaphores are tested in the keyword order SDB1 to SDB4.

The function returns the following values:

Value	Meaning
0	Request timed out
1	Request satisfied by SDB1
2	Request satisfied by SDB2
3	Request satisfied by SDB3
4	Request satisfied by SDB4
5	Error condition

If any of the semaphores has a packet at the time of the call, the function performs a basic receive operation on the first such semaphore encountered, copying message data to user space as requested, and returns to the caller with a value between 1 and 4 to indicate which queue semaphore specified in the call satisfied the request. The packet is dequeued and returned to the free pool as part of the operation.

If none of the semaphores has a packet and either no timeout-interval or a nonzero timeout interval was supplied in the call, the function switches the calling process to the wait-active state. In this state, the process is blocked on all the semaphores specified in the request.

If none of the semaphores has a packet and a zero timeout-interval was supplied in the call, the function returns immediately to the caller with a zero value, indicating a return due to timeout. (The calling process thus never leaves the run state in the case of an immediate timeout.)

If the calling process switches to the wait-active state, the process is blocked from execution until it can be reactivated either by a packet becoming available on one of the blocking semaphores (see SEND or COND_SEND request semantics) or by elapse of the specified timeout period, if any. When reactivated for either reason, the process is unblocked from all the semaphores and is switched to either the ready-active or run state, depending on relative process priorities. If unblocked because of an available packet, the function returns the ordinal value (from 1 to

4) of the semaphore that triggered the return, as described above. If unblocked because of a timeout, the function returns a 0.

If an error occurs, the function returns a 5.

This request is implemented through the RCVA$ kernel primitive.

Error Returns

See Section 11.2 for general information about error returns. The following exception code may be returned:

ES$IST	(type: SYSTEM_SERVICE)—Invalid structure descriptor; no such queue semaphore exists

The request may return the following errors, though not as a result of standard Pascal programming practice:

ES$IAD	(type: SYSTEM_SERVICE)—Invalid address; pointer to buffer, rtnptr is not a word-address (even) value
ES$IPM	(type: SYSTEM_SERVICE)—Illegal parameter; timeout value out of range

Implementation Notes

Since the initial test of the semaphores for an available packet is performed in determinate order, the order in which multiple semaphores are specified in the call can be critical under certain real-time conditions. For example, assume that the relative frequency of signals or sends is high for one of several queue semaphores and that the "fast" queue semaphore is identified as the first to be tested for signals by being associated with keyword SDB1.

In a series of calls to RECEIVE_ANY, that semaphore will be serviced far more often than the others, and the "slower" semaphores may seldom or never be tested and serviced. Optimally, then, the semaphore with the highest expected signal rate should be assigned to the keyword that is tested last; the next highest as next to last, and so on, assuming that probable relative frequencies can be determined. Alternatively, the order in which the semaphores are identified could be rotated in successive calls so at least N semaphores are guaranteed to be tested in N calls to RECEIVE_ANY. The correct or best-case strategy depends on application-specific factors, of course.

14.23 RECEIVE_ANY_ACK

MACRO equivalent: RCVA$

The RECEIVE_ANY_ACK function implements a complex form of the RECEIVE_ACK operation; see the RECEIVE_ACK and SEND_ACK requests for a description of the basic receive acknowledge and send acknowledge message operations on queue semaphores. RECEIVE_ANY_ACK performs the basic RECEIVE_ACK operation on the logical OR of several queue semaphores, with an optional timeout feature. That is, RECEIVE_ANY_ACK permits the calling process to test for and, if necessary, wait on message data on any one of a set of queue semaphores. Up to four queue semaphores can be specified in the request.

The function returns ordinal values from 0 to 5 to indicate the results of the operation (see Semantics).

If no message packet is available on any of the specified semaphores, the calling process blocks until any one of those semaphores is signaled (or sent to) and provides a message for the calling process. (The caller could be blocked behind other waiting processes on a given queue semaphore, of course, although a multiple-receiver policy is unlikely, particularly in the case of RECEIVE_ANY_ACK usage.) The caller receives message data by reference, as described for the basic RECEIVE_ACK operation.

Optionally, a RECEIVE_ANY_ACK operation can be terminated due to the expiration of a time interval specified in the request.

Thus, RECEIVE_ANY_ACK allows a process to get a message from any of up to four queue semaphores, each semaphore being signaled (put or sent to) by a separate process, for example. The function might also be used primarily for its timeout capability.

If a zero time period (immediate timeout) is specified in the request, RECEIVE_ANY_ACK provides a complex form of the COND_RECEIVE_ACK request, which tests for an available message but will not block the caller.

The packet format expected by RECEIVE_ANY_ACK is the same as that produced by the SEND_ACK and COND_SEND_ACK requests, as described in Figure 14–3.

Note

The message-by-reference feature must be used with caution concerning the length of the message. Once the message-copying operation begins, no other process can gain control until the entire message is copied, because of the indivisible nature of kernel primitive operations. Thus, transmission of long messages can seriously affect the servicing of interrupts by increasing interrupt latency throughout the system.

RECEIVE_ANY_ACK is intended for use by processes with general or DEV_ACCESS mapping, which cannot access a packet directly in a mapped-memory environment (see Sections 10.1.2 and 14.1 for more information). This function permits a process, regardless of mapping type, to obtain data from another process through a packet.

The SEND_ACK and COND_SEND_ACK requests permit a process to transmit data by reference and a reply semaphore through a packet.

See COND_RECEIVE_ACK for a description of the basic conditional receive with acknowledgment operation.

Syntax

```
RECEIVE_ANY_ACK ( [[ REPLY_DESC := reply-sem-descriptor ]]
                  REC_LENGTH := ref-xmit-length
                  [[ SDB4 := queue-sem-descriptor-4 ]]
                  [[ SDB3 := queue-sem-descriptor-3 ]]
                  [[ SDB2 := queue-sem-descriptor-2 ]]
                  SDB1 := queue-sem-descriptor-1
                  [[ REF_DATA := reference-data-id
                  REF_LENGTH := reference-data-length ]]
                  [[ TIMEOUT := timeout-interval ]]
                  [[ STATUS := status-record ]] )
```

reply-sem-descriptor

The identifier of a variable of predefined type SEMAPHORE_DESC that will receive the structure identifier of the binary or counting reply semaphore provided by the sender. If you do not specify this parameter, the procedure automatically signals the reply semaphore when the receive operation is complete. Otherwise, the reply semaphore may be signaled manually at the receiver's discretion.

ref-xmit-length

The identifier of a variable of predefined type REF_DATA_LEN that will receive a value that is the number of bytes of data that were sent by reference. This value may be greater than the number of bytes received, which is limited by the reference-data-length parameter. Zero indicates that no data was sent by reference.

queue-sem-descriptor-4
queue-sem-descriptor-3
queue-sem-descriptor-2
queue-sem-descriptor-1

The identifier of a variable of predefined type QUEUE_SEMAPHORE_DESC that contains a semaphore's structure identifier. You can specify up to four queue semaphores. The order in which you specify multiple queue semaphores determines the order in which they are initially tested for a signal. (That order can be critical under certain real-time conditions, as discussed under Semantics and Implementation Notes.)

Each descriptor must have been previously initialized by an INIT_STRUCTURE_DESC or a CREATE_QUEUE_SEMAPHORE request.

reference-data-id

The identifier of the variable (buffer) that will receive the data being sent by reference. This parameter is significant only if the reference-data-length parameter is nonzero.

reference-data-length

A constant or the identifier of a variable of type REF_DATA_LEN that specifies the length, in bytes, of the message buffer (reference-data-id parameter) that will receive the data sent by reference. This parameter limits the amount of data to be copied from the sender's buffer. If the value is 0 and if a message by reference exists in the packet, the message is not copied. The maximum value is 8128 bytes.

timeout-interval

The identifier of a variable of predefined type LONG_INTEGER that specifies the maximum time, in milliseconds, that the caller wishes to be blocked waiting for data. The value must be a positive integer from 0 to (2**31) −1. A value of 0 causes the request to time out immediately if no packet is available on any of the specified semaphores when RECEIVE_ANY_ACK is called. That is, the calling process will never block if the specified time interval is 0. If you do not specify this parameter, the function assumes no timeout for the operation; the calling process may block indefinitely.

status-record

The identifier of a variable of predefined record type EXC_STATUS that may receive an exception type and code. If you specify this parameter, the exception status, either success or error, that results from issuing the request is reported in this variable. Otherwise, an error causes the corresponding exception to be reported. The format of the exception record is described in Section 11.1.2.

Restrictions

- The maximum value for the reference-data-length parameter is 8128.
- The timeout-interval value is limited to a 31-bit positive integer; that is, the sign bit of the high-order word must not be set. (The maximum valid value, in milliseconds, permits a timeout period of just over 24.89 days; see the SLEEP procedure for more detail.)
- If you wish to use fewer than four queue semaphore descriptor parameters, you must assign them beginning with keyword SDB1. You may not assign keyword parameters with higher-numbered suffixes unless all keywords with lower-numbered suffixes are assigned. For example, if the parameter sequence specifies keyword SDB3, the sequence must also include keywords SDB2 and SDB1.

Example

```
%INCLUDE 'COMPLX.PAS'

VAR
  Queue_1, Queue_2, Queue_3 : QUEUE_SEMAPHORE_DESC;
  Buffer : PACKED ARRAY [0..511] OF CHAR;
  Length : 0..512;
  Received : REF_DATA_LEN;
  Reply : SEMAPHORE_DESC;
  Which_one : COMPLEX_FUNC_VALUE;

[PRIORITY(10), STACK_SIZE(100), NAME ('CONSUM')] PROCESS Consumer;
BEGIN

  (* Receive data with acknowledgment from one of three *)
  (* queue semaphores. *)
  Which_one := RECEIVE_ANY_ACK
                (REF_DATA := Buffer,
                 REF_LENGTH := Length,
                 REPLY_DESC := Reply,
                 REC_LENGTH := Received,
                 SDB1 := Queue_1,
                 SDB2 := Queue_2,
                 SDB3 := Queue_3);
  (* Signal the reply semaphore. *)
  SIGNAL (DESC := Reply);

END; (* Process Consumer *)
```

Semantics

The RECEIVE_ANY_ACK function tests each of the queue semaphores specified in the request for an available queue element, or message packet. The queue semaphores are tested in the keyword order SDB1 to SDB4.

The function returns the following values:

Value	Meaning
0	Request timed out
1	Request satisfied by SDB1
2	Request satisfied by SDB2
3	Request satisfied by SDB3
4	Request satisfied by SDB4
5	Error condition

If any of the semaphores has a packet at the time of the call, the function performs a basic receive acknowledge operation on the first such semaphore encountered, copying message data to user space, signaling a reply semaphore as requested. The function returns control to the caller, with a value between 1 and 4 to indicate which queue semaphore specified in the call satisfied the request. The packet pointer is dequeued and returned to the free pool as part of the operation.

If none of the semaphores has a packet and either no timeout argument or a nonzero timeout value was supplied in the call, the function switches the calling process to the wait-active state. In that state, the process is blocked on all the semaphores specified in the request.

If none of the semaphores has a packet and a zero timeout value was supplied in the call, the function returns immediately to the caller with a 0 value, indicating a return due to timeout. (The calling process thus never leaves the run state in the case of an immediate timeout.)

If the calling process switches to the wait-active state, the process is blocked from execution until it can be reactivated either by a packet becoming available on one of the blocking semaphores (see SEND_ACK or COND_SEND_ACK request semantics) or by elapse of the specified timeout period, if any. When reactivated for either reason, the process is unblocked from all the semaphores and is switched to either the ready-active or run state, depending on relative process priorities. If unblocked because of an available packet, the function returns the ordinal value (from 1 to 4) of the semaphore that triggered the return, as described above. If unblocked because of a timeout, the function returns a 0.

If an error occurs, the function returns a 5.

This request is implemented through the RCVA$ kernel primitive.

Error Returns

See Section 11.2 for general information about error returns. The following exception code may be returned:

ES$IST	(type: SYSTEM_SERVICE)—Invalid structure descriptor; no such queue semaphore exists

The request may return the following errors, though not as a result of standard Pascal programming practice:

ES$IAD	(type: SYSTEM_SERVICE)—Invalid address; pointer to buffer, rtnptr is not a word-address (even) value
ES$IPM	(type: SYSTEM_SERVICE)—Illegal parameter; timeout value out of range

Implementation Notes

Since the initial test of the semaphores for an available packet is performed in determinate order, the order in which multiple semaphores are identified in the call can be critical under certain real-time conditions. For example, assume that the relative frequency of signals or sends is high for one of several queue semaphores and that the "fast" queue semaphore is identified as the first to be tested for signals by being associated with keyword SDB1.

In a series of calls to RECEIVE_ANY_ACK, that semaphore will be serviced far more often than the others, and the "slower" semaphores may seldom or never be tested and serviced. Optimally, then, the semaphore with the highest expected signal rate should be assigned to the keyword that is tested last; the next highest as next to last, and so on, assuming that probable relative frequencies can be determined. Alternatively, the order in which the semaphores are identified could be rotated in successive calls so at least N semaphores are guaranteed to be tested in N calls to RECEIVE_ANY_ACK. The correct or best-case strategy depends on application-specific factors, of course.

14.24 SEND

MACRO equivalent: SEND$

The SEND procedure allocates a packet in system space, copies user data into the packet, and signals a specified queue semaphore. SEND provides you with two methods for sending data: by value and by reference. Data sent by value is transmitted in the packet; data sent by reference is copied from a variable specified by the sender to a variable specified by the receiver.

Up to 34 bytes of data can be sent by value; that is, a short message can be sent directly in the packet. A larger amount of data can be sent by reference, or indirectly; a reference to the data, not the data itself, is sent in the packet. You can combine those two methods in one SEND request, sending some data by value and some by reference, or you can use them separately.

The message-by-reference feature permits messages that are too large to fit into a packet to be exchanged between two processes with one SEND and one RECEIVE request. The physical buffer address and length are placed in the packet for subsequent use by the RECEIVE and COND_RECEIVE requests. The actual message is copied from the sender's buffer to the receiver's buffer only when the corresponding receive request is issued.

Note

The message-by-reference feature must be used with caution concerning the length of messages. Once the message-copying operation begins, no other process can gain control until the entire message is copied, because of the indivisible nature of kernel primitive operations. Thus, transmission of long messages can seriously affect the servicing of interrupts by increasing interrupt latency throughout the system.

SEND is intended for use by processes with general or DEV_ACCESS mapping, which cannot access a packet directly in a mapped-memory environment (see Section 10.1.2). This function permits a process, regardless of mapping type, to transmit data to another process through a packet.

The RECEIVE and COND_RECEIVE requests permit a process to receive data sent through a packet.

The COND_SEND request is a conditional form of the SEND request in which a packet of data is placed in the semaphore's queue only when a process is waiting on that queue.

Syntax

```
SEND ( [[ REF_DATA := reference-data-id
       REF_LENGTH := reference-data-length ]]
       [[ VAL_DATA := value-data-id
       VAL_LENGTH := value-data-length ]]
       [[ PRIORITY := packet-priority ]]
       { DESC := queue-sem-descriptor }
       { NAME := queue-sem-name       }
       [[ STATUS := status-record ]] )
```

reference-data-id

The identifier of the variable (buffer) that contains the data to be sent by reference. The address of this variable is converted to a physical address and is placed in the packet with the reference-data-length parameter value. This parameter is significant only if the reference-data-length parameter is nonzero.

reference-data-length

A constant or the identifier of a variable of predefined type REF_DATA_LEN that specifies the length, in bytes, of the message buffer (reference-data-id parameter) containing the data to be sent by reference. The maximum value is 8128 bytes.

value-data-id

The identifier of a variable (buffer) that contains the data to be transmitted by value. The contents of this buffer are copied into the packet directly. This parameter is significant only if the value-data-length parameter is nonzero.

value-data-length

A constant or the identifier of a variable of predefined type VAL_DATA_LEN that specifies the number of bytes to be transmitted by value. The value of this parameter can be from 0 to 34. (See Restrictions.) The default value is 0.

packet-priority

A constant or the identifier of a variable of predefined type PRIORITY_RANGE that specifies the priority value (0 to 255) to be assigned to the packet. This value affects the order in which the packet is queued on a semaphore having a priority-ordered packet queue (see the CREATE_QUEUE_SEMAPHORE request). The default priority value is 1.

queue-sem-descriptor

The identifier of a variable of predefined type QUEUE_SEMAPHORE_DESC that contains the semaphore's structure identifier. The variable must have been previously initialized by a CREATE_QUEUE_SEMAPHORE or an INIT_STRUCTURE_DESC request.

queue-sem-name

A character-string constant or the identifier of a variable of predefined type NAME_STR that specifies the 6-character name of an existing semaphore (see Section 11.1.1.1).

status-record

The identifier of a variable of predefined record type EXC_STATUS that may receive an exception type and code. If you specify this parameter, the exception status, either success or error, that results from issuing the request is reported in this variable. Otherwise, an error causes the corresponding exception to be reported. The format of the exception record is described in Section 11.1.2.

You must specify at least one set of parameters, either value or reference.

Restrictions

- The maximum value for the reference-data-length parameter is 8128.
- A total of 34 bytes is available in a packet for message data, a message reference, or both. A reference occupies three words in the packet and, if included, reduces the space available for data by value to 28 bytes.

Example

```
VAR
  Queue_1 : QUEUE_SEMAPHORE_DESC;
  Buffer : PACKED ARRAY [0..511] OF CHAR;

[PRIORITY(10), STACK_SIZE(100), NAME ('PRODUC')] PROCESS Producer;
BEGIN

  (* Send data by reference. *)
  SEND (REF_DATA := Buffer,
        REF_LENGTH := 512,
        DESC := Queue_1);

  (* Send data by value. *)
  SEND (VAL_DATA := Buffer,
        VAL_LENGTH := 12,
        DESC := Queue_1);

END; (* Process Producer *)
```

Semantics

The SEND procedure performs the following actions prior to signaling the specified queue semaphore:

1. Obtains a packet from the system's free-element pool and writes the specified priority value into the packet header.
2. Constructs a control byte based on the value-data-length and reference-data-length parameters and places it in the packet header for subsequent use by the RECEIVE, RECEIVE_ANY, and COND_RECEIVE requests.
3. Copies the data, if any, to be transmitted by value from the buffer in user space to the packet in system space.
4. In a mapped-memory environment, constructs a physical address from the address (reference-data-id) of the message to be sent by reference, if any. This physical address is placed in the packet along with the message length. A physical address consists of two words. The value of the first word is the virtual address; the value of the second word is the content of the user-mode PAR associated with that virtual address.

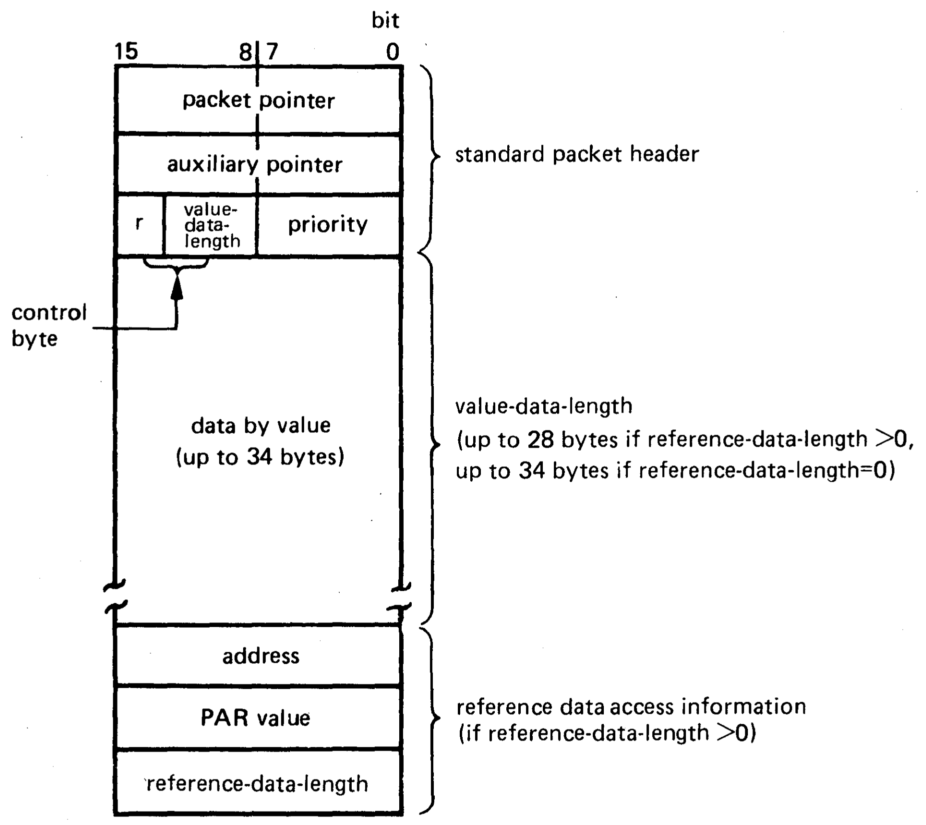
The SEND procedure then tests the specified queue semaphore for waiting processes. If no process is waiting, SEND signals the semaphore, links the packet into the queue, and returns to the caller.

If at least one process is waiting, SEND unblocks the first waiting process, associates the passed packet pointer with that process as its wait-return value, and calls the scheduler, if required. This procedure may cause the calling process to be preempted; to lose control of the CPU.

The format of a packet constructed by SEND (or by COND_SEND) is shown in Figure 14–2.

This request is implemented through the SEND$ kernel primitive.

Figure 14–2: SEND Request Packet Format



MLO–564–87

Error Returns

See Section 11.2 for general information about error returns. The following exception codes may be returned:

ES$CDN	(type: SYSTEM_SERVICE)—Cannot specify both descriptor and name
ES$IST	(type: SYSTEM_SERVICE)—Illegal structure descriptor; no such queue semaphore exists
ES$MDN	(type: SYSTEM_SERVICE)—Must specify descriptor or name

The request may return the following errors, though not as a result of standard Pascal programming practice:

ES$IAD	(type: SYSTEM_SERVICE)—Invalid address; pointer to buffer or structure is odd or not in user address space
ES$IPM	(type: SYSTEM_SERVICE)—Illegal parameter; amount of data to be sent by value (primitive's vlen parameter) exceeds packet capacity or amount of data sent by reference (primitive's rlen parameter) exceeds 8128 bytes

Applications

SEND, the basic buffer transfer service provided by the kernel, provides a message-exchange mechanism for use between general processes or between general and system processes. For example, SEND implements the interface to higher-level services such as those provided by the device-handler processes. This interface consists of a request message sent to the appropriate system process and a reply received from the process, using the SEND and RECEIVE requests.

14.25 SEND_ACK

MACRO equivalent: SEND$

The SEND_ACK procedure allocates a packet in system space, copies a message-by-reference buffer and a reply semaphore descriptor into the packet, and signals a specified queue semaphore. The reply semaphore allows the receiver process to signal an acknowledgment to the sender.

The message-by-reference feature permits messages that are too large to fit into a packet to be exchanged between two processes with one SEND and one RECEIVE request. The physical buffer address and length are placed in the packet for subsequent use by the RECEIVE_ACK and COND_RECEIVE_ACK requests. The message is copied from the sender's buffer to the receiver's buffer only when the corresponding RECEIVE_ACK or RECEIVE_ANY_ACK requests are issued.

Note

> The message-by-reference feature must be used with caution concerning the length of messages. Once the message-copying operation begins, no other process can gain control until the entire message is copied, because of the indivisible nature of kernel primitive operations. Thus, transmission of long messages can seriously affect the servicing of interrupts by increasing interrupt latency throughout the system.

SEND_ACK is intended for use by processes with general or DEV_ACCESS mapping, which cannot access a packet directly in a mapped-memory environment (see Section 10.1.2). This function permits a process, regardless of mapping type, to transmit data by reference and a reply semaphore's structure descriptor to another process through a packet.

The RECEIVE_ACK and COND_RECEIVE_ACK requests allow any process to receive the reference data and reply semaphore transmitted by SEND_ACK.

The COND_SEND_ACK request is a conditional form of SEND_ACK in which a packet of data is placed in the semaphore's queue only when a process is waiting on that queue.

Syntax

```
SEND_ACK ( [[ REF_DATA := reference-data-id
            REF_LENGTH := reference-data-length ]]
            REPLY_DESC := reply-sem-descriptor
            [[ PRIORITY := packet-priority ]]
            { DESC := queue-sem-descriptor }
            { NAME := queue-sem-name       }
            [[ STATUS := status-record ]] )
```

reference-data-id

The identifier of the variable (buffer) that contains the data to be sent by reference. The address of this variable is converted to a physical address and is placed in the packet with the reference-data-length parameter value. This parameter is significant only if the reference-data-length parameter is nonzero.

reference-data-length

A constant or the identifier of a variable of predefined type REF_DATA_LEN that specifies the length, in bytes, of the message buffer (reference-data-id parameter) containing the data to be sent by reference. The maximum value is 8128 bytes.

reply-sem-descriptor

The identifier of a variable of predefined type SEMAPHORE_DESC that contains the reply semaphore's structure identifier. The variable must have been previously initialized by a CREATE_BINARY_SEMAPHORE or a CREATE_COUNTING_SEMAPHORE, as appropriate, or by an INIT_STRUCTURE_DESC request.

packet-priority

A constant or the identifier of a variable of predefined type PRIORITY_RANGE that specifies the priority value (0 to 255) to be assigned to the packet. This value affects the order in which the packet is queued on a semaphore having a priority-ordered packet queue (see the CREATE_QUEUE_SEMAPHORE request). The default priority value is 1.

queue-sem-descriptor

The identifier of a variable of predefined type QUEUE_SEMAPHORE_DESC that contains the semaphore's structure identifier. The variable must have been previously initialized by a CREATE_QUEUE_SEMAPHORE or an INIT_STRUCTURE_DESC request.

queue-sem-name

A character-string constant or the identifier of a variable of predefined type NAME_STR that specifies the 6-character name of an existing semaphore (see Section 11.1.1.1).

status-record

The identifier of a variable of predefined record type EXC_STATUS that may receive an exception type and code. If you specify this parameter, the exception status, either success or error, that results from issuing the request is reported in this variable. Otherwise, an error causes the corresponding exception to be reported. The format of the exception record is described in Section 11.1.2.

Example

```
VAR
  Queue_1 : QUEUE_SEMAPHORE_DESC;
  Buffer : PACKED ARRAY [0..511] OF CHAR;
  Reply : SEMAPHORE_DESC;

[PRIORITY(10), STACK_SIZE(100), NAME ('PRODUC')] PROCESS Producer;
BEGIN

  (* Send data with acknowledgment. *)
  SEND_ACK (REF_DATA := Buffer,
            REF_LENGTH := 512,
            REPLY_DESC := Reply,
            DESC := Queue_1);

  (* Wait for a reply. *)
  WAIT (DESC := Reply);

END; (* Process Producer *)
```

Restriction

The maximum value for the reference-data-length parameter is 8128.

Semantics

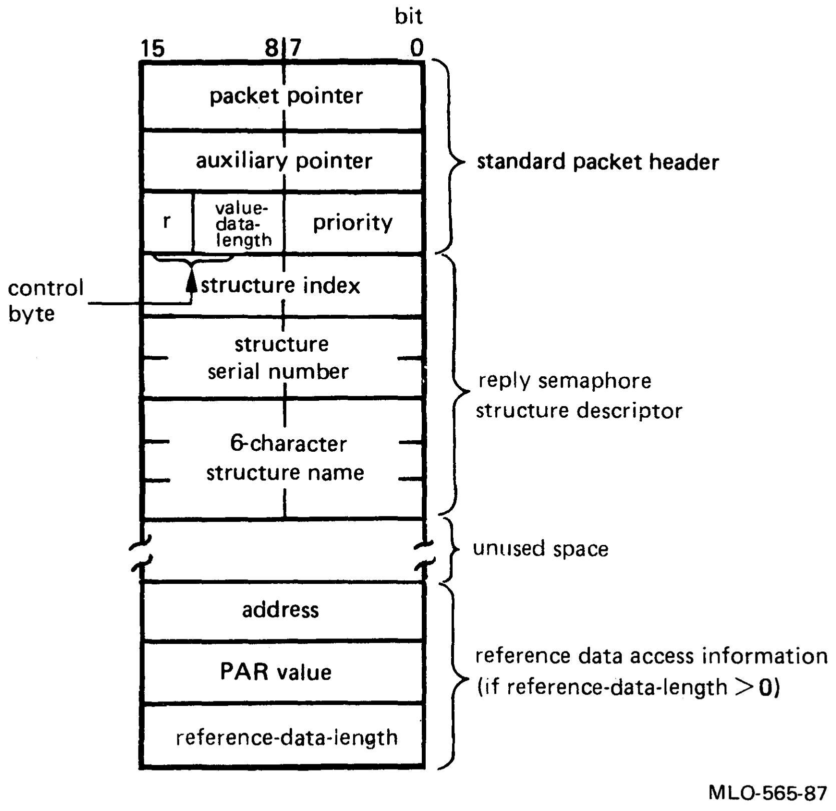
The SEND_ACK procedure performs the following actions prior to signaling the specified queue semaphore:

1. Obtains a packet from the system's free-element pool and writes the specified priority value into the packet header.
2. Constructs a control byte that consists of the reference data flag bit (r) and a 7-bit value-data-length field containing the length of the structure identifier used for the reply semaphore.
3. Places the control byte in the packet header for subsequent use by the RECEIVE_ACK, RECEIVE_ANY_ACK, and COND_RECEIVE_ACK requests and copies the reply semaphore structure identifier into the packet's value data area.
4. Constructs a physical address from the address of the message to be sent by reference, if any. This physical address is placed in the packet along with the message length. A physical address consists of two words. The value of the first word is the virtual address; the value of the second word is the content of the user-mode PAR associated with that virtual address.

SEND_ACK then tests the specified queue semaphore for waiting processes. If no process is waiting, SEND_ACK signals the semaphore, links the packet into the queue, and returns to the caller. If at least one process is waiting, this procedure unblocks the first waiting process, associates the passed packet pointer with that process as its wait-return value, and calls the scheduler, which may cause the calling process to be preempted; to lose control of the CPU.

The format of a packet constructed by SEND_ACK (or by COND_SEND_ACK) is shown in Figure 14–3.

Figure 14-3: SEND_ACK Request Packet Format



This request is implemented through the SEND$ kernel primitive.

Error Returns

See Section 11.2 for general information about error returns. The following exception codes may be returned:

ES$CDN	(type: SYSTEM_SERVICE)—Cannot specify both descriptor and name
ES$IST	(type: SYSTEM_SERVICE)—Illegal structure descriptor; no such queue semaphore exists
ES$MDN	(type: SYSTEM_SERVICE)—Must specify descriptor or name

The request may return the following errors, though not as a result of standard Pascal programming practice:

ES$IAD	(type: SYSTEM_SERVICE)—Invalid address; pointer to buffer or structure is odd or not in user address space
ES$IPM	(type: SYSTEM_SERVICE)—Illegal parameter; amount of data to be sent by value (primitive's vlen parameter) exceeds packet capacity or amount of data sent by reference (primitive's rlen parameter) exceeds 8128 bytes

Chapter 15
Ring Buffer Management Requests

This chapter describes the requests that operate on ring buffer structures. Those requests, implemented through the predeclared procedures and functions listed in Table 15–1, are the Pascal language interface to the services provided by the kernel's ring buffer management primitives. The requests provide for variable-length data transfers (normally a stream of byte data) between processes, without the need for tight, signal/wait synchronization between them. The size, or capacity, of a ring buffer is determined when the structure is created and may be from 8 to 8128 bytes.

You must include in your program or module the definitions of the routines that implement the requests described in this chapter before using them. See Appendix I for more information.

Table 15–1 summarizes the operations performed by these requests.

Table 15–1: Ring Buffer Management Requests

Request	Operation
COND_GET_ELEMENT	A conditional form of GET_ELEMENT that does not block the calling process if the request cannot be satisfied.
COND_PUT_ELEMENT	A conditional form of PUT_ELEMENT that does not block the calling process if the request cannot be satisfied.
CREATE_RING_BUFFER	Creates a ring buffer structure and sets up a descriptor for efficient reference to it.
CREATE_RING_BUFFER_P	Creates a ring buffer structure and sets up a descriptor for efficient reference to it by a procedure.
DESTROY	Deletes a structure from the system and deallocates the memory space used by it.

Table 15-1 (Cont.): Ring Buffer Management Requests

Request	Operation
GET_ELEMENT	Extracts a specified number of bytes of data from a ring buffer and transfers them to the caller's buffer; the calling process is blocked if too few bytes are in the ring buffer.
GET_ELEMENT_ANY	An enhanced form of GET_ELEMENT that waits for a variable time interval on up to four ring buffers.
GET_VALUE	Obtains a structure's value and type code.
INIT_STRUCTURE_DESC	Sets up a descriptor for efficient reference to a ring buffer.
PUT_ELEMENT	Copies a specified number of bytes from the caller's buffer to a ring buffer; the calling process is blocked if the ring buffer has insufficient space.
RESET_RING_BUFFER	Empties the specified ring buffer of all data.

15.1 COND_GET_ELEMENT

MACRO equivalent: GELC$

The COND_GET_ELEMENT function implements a nonblocking form of the GET_ELEMENT request. The function attempts to copy the requested number of bytes of data from the ring buffer but does not block the caller if the request cannot be satisfied. The output access mode of the ring buffer (record mode or stream mode) determines whether the request attempts to satisfy the data transfer by a full or by a partial transfer. Informally, the meaning of a COND_GET_ELEMENT request for a record-mode buffer is "get N bytes right away or none at all" and for a stream-mode buffer is "get as many bytes as possible, up to N right away."

In either case, however, the function returns control to the caller, with a value that indicates how many bytes are needed to satisfy the request. A return value of 0 indicates that the request has been fully satisfied—all the bytes specified in the call have been successfully read from the ring buffer.

The output access mode of a ring buffer is declared to be either RECORD_MODE or STREAM_MODE when the structure is created. (See the CREATE_RING_BUFFER request.)

For a ring buffer in record mode, COND_GET_ELEMENT attempts to satisfy the request with a full transfer only. If the ring buffer does not contain as many bytes as are requested, the function returns control immediately to the caller, with a value that is the number of bytes requested, thus indicating that no bytes were copied.

For a ring buffer in stream mode, the default, COND_GET_ELEMENT attempts to satisfy the request with either a full or a partial transfer. That is, the function obtains as many bytes, up to the number requested, as are available in the ring buffer. The function returns control to the caller, with a value that is the number of bytes, if any, that remain to be obtained.

The PUT_ELEMENT and COND_PUT_ELEMENT requests allow a process to insert bytes into a ring buffer.

Syntax

```
COND_GET_ELEMENT ( LENGTH := data-length
                   DATA := data-id
                   { DESC := ring-buffer-descriptor }
                   { NAME := ring-buffer-name       }
                   [[ STATUS := status-record ]] )
```

data-length

A constant or the identifier of a variable of type RING_BUFFER_DATA that specifies the number of bytes of data to be read from the ring buffer. This value is the length of the data identified by the data-id parameter.

data-id

The identifier of the variable (buffer) that is to receive the data being read from the ring buffer.

ring-buffer-descriptor

The identifier of a variable of predefined type RING_BUFFER_DESC that contains the ring buffer's structure identifier. The variable must have been previously initialized by a CREATE_RING_BUFFER or an INIT_STRUCTURE_DESC request.

ring-buffer-name

A character-string constant or the identifier of a variable of predefined type NAME_STR that contains the 6-character name of an existing ring buffer (see Section 11.1.1.1).

status-record

The identifier of a variable of predefined record type EXC_STATUS that may receive an exception type and code. If you specify this parameter, the exception status, either success or error, that results from issuing the request is reported in this variable. Otherwise, an error causes the corresponding exception to be reported. The format of the exception record is described in Section 11.1.2.

Restriction

If the ring buffer's output access mode is record mode, the number of bytes specified by the data-length parameter must not exceed the size of the ring buffer (as specified in the CREATE_RING_BUFFER request); otherwise, the request will never be successful.

Example

```
VAR
  Ring_1 : RING_BUFFER_DESC;
  Ch : CHAR;
  Short : INTEGER;

[PRIORITY(10), STACK_SIZE(100), NAME ('READER')] PROCESS Reader;
BEGIN

  (* Conditionally get a character from an unnamed ring buffer. *)
  Short := COND_GET_ELEMENT (LENGTH := 1,
                             DATA := Ch,
                             DESC := Ring_1);

  (* Conditionally get a character from a named ring buffer. *)
  Short := COND_GET_ELEMENT (LENGTH := 1,
                             DATA := Ch,
                             NAME := 'RING1 ');

END; (* Process Reader *)
```

Semantics

If the specified ring buffer's output access mode is record mode, the COND_GET_ELEMENT function tests the ring buffer. If the ring buffer contains at least that number of bytes, the function transfers the specified number of bytes of data from the ring buffer to the caller's storage area and returns control to the caller, with the value 0. If the ring buffer contains less than the requested number of bytes, the function returns control immediately to the caller, with the original value of the data-length parameter to indicate that no bytes were transferred.

If the ring buffer's output access mode is stream mode, the function copies as many bytes, up to the number requested, from the ring buffer to the caller's storage area and returns control to the caller, with the value that is the number of bytes that remain to be copied (data-length minus bytes transferred).

Note

> A successful COND_GET_ELEMENT operation may cause preemption of the caller if the operation unblocks a process waiting to put elements. That is, return from a successful COND_GET_ELEMENT request is not necessarily immediate.

This request is implemented through the GELC$ kernel primitive.

Error Returns

See Section 11.2 for general information about error returns. The following exception codes may be returned:

ES$CDN	(type: SYSTEM_SERVICE)—Cannot specify both descriptor and name
ES$IST	(type: SYSTEM_SERVICE)—Illegal structure descriptor; no such ring buffer exists
ES$MDN	(type: SYSTEM_SERVICE)—Must specify descriptor or name

The request may return the following error, though not as a result of standard Pascal programming practice:

ES$IAD	(type: SYSTEM_SERVICE)—Invalid address; pointer to buffer or structure is odd or not in user address space

15.2 COND_PUT_ELEMENT

MACRO equivalent: PELC$

The COND_PUT_ELEMENT function implements a nonblocking form of the PUT_ELEMENT request. The function attempts to copy the requested number of bytes of data from the caller's storage area to a ring buffer but does not block the calling process if the request cannot be satisfied. The input access mode of the ring buffer (record mode or stream mode) determines how the request attempts to satisfy the data transfer. In either case, however, the function returns control to the caller, with a value that indicates how many bytes remain to be transferred. A return value of 0 indicates that the request has been fully satisfied; all the bytes specified in the call have been successfully put into the ring buffer.

The input access mode of a ring buffer is declared to be either RECORD_MODE or STREAM_MODE when the structure is created. (See the CREATE_RING_BUFFER request.)

For a ring buffer in record mode, COND_PUT_ELEMENT attempts to satisfy the request with a full transfer only. If the ring buffer has insufficient space for the specified number of bytes, it returns control immediately to the caller, with a value equal to the number of bytes specified in the request, indicating that no bytes were copied.

For a ring buffer in stream mode, the default, COND_PUT_ELEMENT attempts to satisfy the request with either a full or a partial transfer. That is, the function inserts all the bytes that can be accommodated in the buffer—none, some, or all those requested—and returns control to the caller, with a value that is the number of bytes, if any, that remain to be transferred.

The GET_ELEMENT and COND_GET_ELEMENT requests allow a process to extract bytes from a ring buffer, freeing the associated space.

Syntax

```
COND_PUT_ELEMENT ( LENGTH := data-length
                   DATA := data-id
                   { DESC := ring-buffer-descriptor }
                   { NAME := ring-buffer-name       }
                   [[ STATUS := status-record ]] )
```

data-length

A constant or the identifier of a variable of type RING_BUFFER_DATA that specifies the number of bytes of data to copy into the ring buffer. This value is the length of the data identified by the data-id parameter.

data-id

The identifier of the variable (buffer) that contains data to be copied into the ring buffer.

ring-buffer-descriptor

The identifier of a variable of predefined type RING_BUFFER_DESC that contains the ring buffer's structure identifier. The variable must have been previously initialized by a CREATE_RING_BUFFER or an INIT_STRUCTURE_DESC request.

ring-buffer-name

A character-string constant or the identifier of a variable of predefined type NAME_STR that contains the 6-character name of an existing ring buffer (see Section 11.1.1.1).

status-record

The identifier of a variable of predefined record type EXC_STATUS that may receive an exception type and code. If you specify this parameter, the exception status, either success or error, that results from issuing the request is reported in this variable. Otherwise, an error causes the corresponding exception to be reported. The format of the exception record is described in Section 11.1.2.

Restriction

If the ring buffer's input access mode is record mode, the number of bytes specified by the data-length parameter must not exceed the size of the ring buffer (as specified in the CREATE_RING_BUFFER request); otherwise, the request will never be successful.

Example

```
VAR
  Ring_1 : RING_BUFFER_DESC;
  Remaining : INTEGER;

[PRIORITY(10), STACK_SIZE(100), NAME ('WRITER')] PROCESS Writer;
BEGIN

  (* Conditionally put a character into an unnamed ring buffer. *)
  Remaining := COND_PUT_ELEMENT (LENGTH := 1,
                                DATA := 'A',
                                DESC := Ring_1);

  (* Conditionally put a character into a named ring buffer. *)
  Remaining := COND_PUT_ELEMENT (LENGTH := 1,
                                DATA := 'A',
                                NAME := 'RING1 ');

END; (* Process Writer *)
```

Semantics

If the ring buffer's input access mode is record mode, the COND_PUT_ELEMENT function tests the ring buffer for an amount of available space equal to or greater than the number of bytes specified in the data-length parameter. If at least that amount of space is available, the function copies the specified number of bytes of data from the caller's buffer to the ring buffer and returns control to the caller, with the value 0. If fewer than the specified number of bytes of space are available, the function returns control immediately to the caller, with the original value of the data-length parameter to indicate that no bytes were copied into the ring buffer.

If the ring buffer's input access mode is stream mode, the function copies as many bytes from the caller's buffer as can be accommodated in the ring buffer and returns control to the caller, with a value that is the number of bytes that remain to be copied—data length minus bytes copied.

This request is implemented through the PELC$ kernel primitive.

Error Returns

See Section 11.2 for general information about error returns. The following exception codes may be returned:

ES$CDN	(type: SYSTEM_SERVICE)—Cannot specify both descriptor and name
ES$IST	(type: SYSTEM_SERVICE)—Illegal structure descriptor; no such ring buffer exists
ES$MDN	(type: SYSTEM_SERVICE)—Must specify descriptor or name

The request may return the following error, though not as a result of standard Pascal programming practice:

ES$IAD	(type: SYSTEM_SERVICE)—Invalid address; pointer to buffer or structure is odd or not in user address space

15.3 CREATE_RING_BUFFER

MACRO equivalent: CRST$

The CREATE_RING_BUFFER function creates a ring buffer structure in the system-common memory managed by the kernel.

If the buffer is successfully created, the request returns a Boolean TRUE value. If system memory is insufficient to create the buffer, the function returns a Boolean FALSE value.

The function permits a process to create a ring buffer that can be manipulated by the various ring buffer management requests.

Syntax

```
CREATE_RING_BUFFER ( [[ INPUT_MODE := { RECORD_MODE } ]]
                                      { STREAM_MODE }
                     [[ OUTPUT_MODE := { RECORD_MODE } ]]
                                       { STREAM_MODE }
                     [[ INPUT_ORDER := { FIFO } ]]
                                       { PRIO }
                     [[ OUTPUT_ORDER := { FIFO } ]]
                                        { PRIO }
                     SIZE := buffer-size
                     { DESC := ring-buffer-descriptor }
                     { NAME := ring-buffer-name       }
                     [[ STATUS := status-record ]] )
```

INPUT_MODE
The input access mode for transferring data to the ring buffer when using the COND_PUT_ELEMENT procedure. RECORD_MODE specifies that a request will be honored only when the ring buffer has enough space to allow a complete transfer of all data. STREAM_MODE, the default, specifies that a request will be honored whenever the buffer has space to allow one or more bytes of the data to be transferred.

OUTPUT_MODE
The output access mode for obtaining data from the ring buffer when using the COND_GET_ELEMENT procedure. RECORD_MODE specifies that a request will be honored only when the ring buffer has enough space to allow a complete transfer of all data. STREAM_MODE, the default, specifies that a request will be honored whenever the buffer has space to allow one or more bytes of the data to be transferred.

INPUT_ORDER
The ordering of the ring buffer's input queue, which contains the list of processes waiting to put bytes into the buffer. FIFO specifies first-in-first-out order and is the default value. PRIO specifies ordering by process priority.

OUTPUT_ORDER
The ordering of the ring buffer's output queue, which contains the list of processes waiting to get bytes from the ring buffer. FIFO specifies first-in-first-out order and is the default value. PRIO specifies ordering by process priority.

buffer-size

A constant or the identifier of a variable of type RING_BUFFER_SIZE that specifies the size, in bytes, of the ring buffer. This value must be an even number from 8 to 8128.

ring-buffer-descriptor

The identifier of a variable of predefined type RING_BUFFER_DESC that contains the ring buffer's structure identifier.

ring-buffer-name

A character-string constant or the identifier of a variable of predefined type NAME_STR that contains the 6-character name of the ring buffer (see Section 11.1.1.1). This must not be the name of an existing process or structure.

status-record

The identifier of a variable of predefined record type EXC_STATUS that may receive an exception type and code. If you specify this parameter, the exception status, either success or error, that results from issuing the request is reported in this variable. Otherwise, an error causes the corresponding exception to be reported. The format of the exception record is described in Section 11.1.2.

If you do not specify the ring-buffer-name parameter, the function creates an unnamed ring buffer identified by the descriptor specified in the ring-buffer-descriptor parameter.

Example

```
%INCLUDE 'EXC.PAS'

VAR
  Ring_1, Ring_2, Ring_3 : RING_BUFFER_DESC;

[INITIALIZE] PROCEDURE Init;
(* Create the needed ring buffers. If any create fails then report an exception. *)
BEGIN
  (* Create an unnamed ring buffer with all of the defaults. *)
  IF NOT CREATE_RING_BUFFER
          (DESC := Ring_1,
          SIZE := 10)
   THEN REPORT
          (EXC_TYPE := [RESOURCE],
          EXC_CODE := ES$NMK,
          EXC_INFO := 0);

  (* Create a named ring buffer with all of the defaults. *)
  IF NOT CREATE_RING_BUFFER
          (DESC := Ring_2,
          SIZE := 10,
          NAME := 'RING2 ')
   THEN REPORT
          (EXC_TYPE := [RESOURCE],
          EXC_CODE := ES$NMK,
          EXC_INFO := 0);
  (* Create an unnamed ring buffer with priority orderings. *)
  IF NOT CREATE_RING_BUFFER
          (INPUT_ORDER := PRIO,
          OUTPUT_ORDER := PRIO,
          DESC := Ring_3,
          SIZE := 10)
   THEN REPORT
          (EXC_TYPE := [RESOURCE],
          EXC_CODE := ES$NMK,
          EXC_INFO := 0);

END; (* Procedure Init *)
```

Semantics

The CREATE_RING_BUFFER function requests the kernel to allocate and initialize a ring buffer structure in system-common memory.

If the ring buffer is successfully created, the function returns a Boolean TRUE value. The ring buffer is named as specified in the ring-buffer-name parameter, and its structure identifier is copied into the descriptor variable specified by the ring-buffer-descriptor parameter.

If the ring buffer cannot be created because of insufficient space in the system's free-memory pool, the function returns control to the caller, with a Boolean FALSE value.

This request is implemented through the CRST$ kernel primitive.

Error Returns

See Section 11.2 for general information about error returns. The following exception codes may be returned:

ES$MDN	(type: SYSTEM_SERVICE)—Must specify structure descriptor or name
ES$SNI	(type: SYSTEM_SERVICE)—Structure name already in use; the specified name has already been given to another process, semaphore, ring buffer, logical name, or shared region

The request may return the following errors, though not as a result of standard Pascal programming practice:

ES$IAD	(type: SYSTEM_SERVICE)—Invalid address; pointer to descriptor is odd or not in user address space
ES$IPM	(type: SYSTEM_SERVICE)—Illegal parameter; an invalid structure type or argument value was specified
ES$IPR	(type: SYSTEM_SERVICE)—Illegal primitive; user code attempted to create a PCB (type ST.PCB specified)

15.4 CREATE_RING_BUFFER_P

MACRO equivalent: none

CREATE_RING_BUFFER_P creates (by a procedure) a ring buffer structure in the system-common memory managed by the kernel.

If the buffer is successfully created, the STATUS parameter is set to ES$NOR. If system memory is insufficient to create the buffer, the STATUS parameter is set to the appropriate exception code.

The procedure permits a process to create a ring buffer that can be manipulated by the various ring buffer management requests.

Syntax

```
CREATE_RING_BUFFER_P (  [[ INPUT_MODE := { RECORD_MODE } ]]
                                         { STREAM_MODE }
                        [[ OUTPUT_MODE := { RECORD_MODE } ]]
                                          { STREAM_MODE }
                        [[ INPUT_ORDER := { FIFO } ]]
                                          { PRIO }
                        [[ OUTPUT_ORDER := { FIFO } ]]
                                           { PRIO }
                        SIZE := buffer-size
                        { DESC := ring-buffer-descriptor }
                        { NAME := ring-buffer-name       }
                        [[ STATUS := status-record ]] )
```

INPUT_MODE

The input access mode for transferring data to the ring buffer when using the COND_PUT_ELEMENT procedure. RECORD_MODE specifies that a request will be honored only when the ring buffer has enough space to allow a complete transfer of all data. STREAM_MODE, the default, specifies that a request will be honored whenever the buffer has space to allow one or more bytes of the data to be transferred.

OUTPUT_MODE

The output access mode for obtaining data from the ring buffer when using the COND_GET_ELEMENT procedure. RECORD_MODE specifies that a request will be honored only when the ring buffer has enough space to allow a complete transfer of all data. STREAM_MODE, the default, specifies that a request will be honored whenever the buffer has space to allow one or more bytes of the data to be transferred.

INPUT_ORDER

The ordering of the ring buffer's input queue, which contains the list of processes waiting to put bytes into the buffer. FIFO specifies first-in-first-out order and is the default value. PRIO specifies ordering by process priority.

OUTPUT_ORDER

The ordering of the ring buffer's output queue, which contains the list of processes waiting to get bytes from the ring buffer. FIFO specifies first-in-first-out order and is the default value. PRIO specifies ordering by process priority.

buffer-size

A constant or the identifier of a variable of type RING_BUFFER_SIZE that specifies the size, in bytes, of the ring buffer. This value must be an even number from 8 to 8128.

ring-buffer-descriptor

The identifier of a variable of predefined type RING_BUFFER_DESC that contains the ring buffer's structure identifier.

ring-buffer-name

A character-string constant or the identifier of a variable of predefined type NAME_STR that contains the 6-character name of the ring buffer (see Section 11.1.1.1). This must not be the name of an existing process or structure.

status-record

The identifier of a variable of predefined record type EXC_STATUS that may receive an exception type and code. If you specify this parameter, the exception status, either success or error, that results from issuing the request is reported in this variable. Otherwise, an error causes the corresponding exception to be reported. The format of the exception record is described in Section 11.1.2.

If you do not specify the ring-buffer-name parameter, the function creates an unnamed ring buffer identified by the descriptor specified in the ring-buffer-descriptor parameter.

Example

```
%INCLUDE 'EXC.PAS'
%INCLUDE 'CRPROC.PAS'

VAR
   RDESC1, RDESC2 : RING_BUFFER_DESC;
   P_STATUS : EXC_STATUS;
   SUCCESS : BOOLEAN;

(* Create the needed ring buffers. If any create fails then set SUCCESS to false. *)
[INITIALIZE] PROCEDURE Init;
BEGIN
   CREATE_RING_BUFFER_P (INPUT_MODE := STREAM_MODE,
                         OUTPUT_MODE := STREAM_MODE,
                         INPUT_ORDER := FIFO, OUTPUT_ORDER := FIFO,
                         BUFFER_SIZE := 8, DESC := DRESC1,
                         NAME := 'RBUFF ', STATUS := P_STATUS);
   IF (P_STATUS.EXC_CODE <> ES$NOR)
                         THEN SUCCESS := False;
   CREATE_RING_BUFFER_P (INPUT_ORDER := PRIO, OUTPUT_ORDER := PRIO,
                         BUFFER_SIZE := 8128, DESC := DRESC2,
                         STATUS := P_STATUS);
   IF (P_STATUS.EXC_CODE <> ES$NOR)
                         THEN SUCCESS := False;
END;
```

```
BEGIN (* Main *)
   IF NOT SUCCESS
      THEN WRITELN('%ERROR - Semaphore creation failed')
      ELSE
      .
      .
      .
END.
```

Semantics

The CREATE_RING_BUFFER_P procedure requests the kernel to allocate and initialize a ring buffer structure in system-common memory.

If the ring buffer is successfully created, the STATUS parameter is set to ES$NOR. The ring buffer is named as specified in the ring-buffer-name parameter, and its structure identifier is copied into the descriptor variable specified by the ring-buffer-descriptor parameter.

If the ring buffer cannot be created because of insufficient space in the system's free-memory pool, the STATUS parameter is set to the appropriate exception code.

This request is implemented through the CRST$ kernel primitive.

Error Returns

See Section 11.2 for general information about error returns. The following exception codes may be returned:

ES$MDN	(type: SYSTEM_SERVICE)—Must specify structure descriptor or name
ES$SNI	(type: SYSTEM_SERVICE)—Structure name already in use; the specified name has already been given to another process, semaphore, ring buffer, logical name, or shared region

The request may return the following errors, though not as a result of standard Pascal programming practice:

ES$IAD	(type: SYSTEM_SERVICE)—Invalid address; pointer to descriptor is odd or not in user address space
ES$IPM	(type: SYSTEM_SERVICE)—Illegal parameter; an invalid structure type or argument value was specified
ES$IPR	(type: SYSTEM_SERVICE)—Illegal primitive; user code attempted to create a PCB (type ST.PCB specified)

15.5 DESTROY

MACRO equivalent: DLST$

The DESTROY procedure deletes a structure (in this case, a ring buffer) from the system and deallocates the memory space associated with it. The operation is performed only if no processes are blocked on the ring buffer at the time of the call.

Syntax

```
DESTROY ( { DESC := ring-buffer-descriptor }
          { NAME := ring-buffer-name       }
          ⟦ STATUS := status-record ⟧ )
```

ring-buffer-descriptor
The identifier of a variable of predefined type RING_BUFFER_DESC that contains the ring buffer's structure identifier. The variable must have been previously initialized by a CREATE_RING_BUFFER or an INIT_STRUCTURE_DESC request.

ring-buffer-name
A character-string constant or the identifier of a variable of predefined type NAME_STR that contains the 6-character name of an existing ring buffer (see Section 11.1.1.1).

status-record
The identifier of a variable of predefined record type EXC_STATUS that may receive an exception type and code. If you specify this parameter, the exception status, either success or error, that results from issuing the request is reported in this variable. Otherwise, an error causes the corresponding exception to be reported. The format of the exception record is described in Section 11.1.2.

Example

```
VAR
  Ring : RING_BUFFER_DESC;

[TERMINATE] PROCEDURE Term;
BEGIN

  (* Destroy an unnamed ring buffer. *)
  DESTROY (DESC := Ring);

  (* Destroy a named ring buffer. *)
  DESTROY (NAME := 'RING  ');

END; (* Procedure Term *)
```

Semantics

If the ring buffer is not in use, DESTROY removes its name, if one exists, from the system name table, returns the space that the ring buffer occupies to the free-memory pool, and returns control to the caller.

This request is implemented through the DLST$ kernel primitive.

Error Returns

See Section 11.2 for general information about error returns. The following exception codes may be returned:

ES$CDN	(type: SYSTEM_SERVICE)—Cannot specify both descriptor and name
ES$IST	(type: SYSTEM_SERVICE)—Illegal structure descriptor; no such ring buffer exists
ES$MDN	(type: SYSTEM_SERVICE)—Must specify descriptor or name
ES$SIU	(type: SYSTEM_SERVICE)—Structure is in use and cannot be deleted

15.6 GET_ELEMENT

MACRO equivalent: GELM$

The GET_ELEMENT procedure copies a specified number of bytes of data from a ring buffer to the caller's storage area. If too few bytes are in the ring buffer to satisfy the request, the calling process waits, or blocks, on the ring buffer for more bytes to become available.

In general, if two or more processes are getting data from the same ring buffer, the calling process will block if another process is waiting for its GET_ELEMENT request to be satisfied. The calling process must wait its turn—whether by FIFO, the default, or by priority order—for access to the buffer, since sequential access to a ring buffer is ensured among multiple readers as well as among multiple writers. If the ring buffer's output access mode is stream, the process that blocks first is given active read access to the buffer and the data transfer may occur in increments while the process is waiting. In stream mode, therefore, the process with active access is never displaced by a higher-priority process, regardless of the ordering attribute of the waiting-output-process list.

The COND_GET_ELEMENT function is the conditional form of the GET_ELEMENT request.

The PUT_ELEMENT and the COND_PUT_ELEMENT requests allow a process to insert bytes into a ring buffer.

Syntax

```
GET_ELEMENT (  LENGTH := data-length
               DATA := data-id
               { DESC := ring-buffer-descriptor }
               { NAME := ring-buffer-name       }
               ⟦ STATUS := status-record ⟧ )
```

data-length

A constant or the identifier of a variable of type RING_BUFFER_DATA that specifies the number of bytes of data to read from the ring buffer. This value is the length of the data identified by the data-id parameter.

data-id

The identifier of the variable (buffer) that is to receive the data being read from the ring buffer.

ring-buffer-descriptor

The identifier of a variable of predefined type RING_BUFFER_DESC that contains the ring buffer's structure identifier. The variable must have been previously initialized by a CREATE_RING_BUFFER or an INIT_STRUCTURE_DESC request.

ring-buffer-name

A character-string constant or the identifier of a variable of predefined type NAME_STR that contains the 6-character name of an existing ring buffer (see Section 11.1.1.1).

status-record

The identifier of a variable of predefined record type EXC_STATUS that may receive an exception type and code. If you specify this parameter, the exception status, either success or error, that results from issuing the request is reported in this variable. Otherwise, an error causes the corresponding exception to be reported. The format of the exception record is described in Section 11.1.2.

Restriction

If the ring buffer's output access mode is record mode, the number of bytes specified by the data-length parameter must not exceed the size of the ring buffer (as specified in the CREATE_RING_BUFFER request); otherwise, the request will never be successful.

Example

```
VAR
  Ring_1 : RING_BUFFER_DESC;
  Ch : CHAR;

[PRIORITY(10), STACK_SIZE(100), NAME ('READER')] PROCESS Reader;
BEGIN

  (* Get a character from an unnamed ring buffer. *)
  GET_ELEMENT (LENGTH := 1,
               DATA := Ch,
               DESC := Ring_1);

  (* Get a character from a named ring buffer. *)
  GET_ELEMENT (LENGTH := 1,
               DATA := Ch,
               NAME := 'RING1 ');

END; (* Process Reader *)
```

Semantics

If no other process is waiting to obtain data from the specified ring buffer, GET_ELEMENT tests the buffer for data-length number of bytes of available data. If at least that amount of data is available, GET_ELEMENT transfers the requested number of bytes from the ring buffer to the caller's buffer and returns control to the caller. (A get operation effectively removes the corresponding data from the ring buffer.)

If the ring buffer contains less than data-length bytes and its output access mode is record mode, GET_ELEMENT blocks the caller with active read access to the ring buffer and calls the scheduler. When a full record becomes available as a result of one or more subsequent PUT_ELEMENT or COND_PUT_ELEMENT operations, the transfer is performed and the waiting process is unblocked.

If the ring buffer contains less than data-length bytes and its output access mode is stream mode, GET_ELEMENT blocks the caller with active read access to the ring buffer, transfers any currently available bytes, and calls the scheduler. When enough additional bytes become available as a result of one or more subsequent PUT_ELEMENT or COND_PUT_ELEMENT operations, the transfer is completed—possibly by a series of partial transfers—and the waiting process is unblocked.

If one or more processes are waiting to get data from the ring buffer at the time of the call, implying that some other process has active read access, the calling process is blocked on the buffer's waiting-output-process list in either FIFO or priority order, depending on the ring buffer definition. (A process with active access is never displaced by another process, regardless of relative priorities.) The process waits its turn to gain active read access, at which point it is treated as described above.

This request is implemented through the GELM$ kernel primitive.

Error Returns

See Section 11.2 for general information about error returns. The following exception codes may be returned:

ES$CDN	(type: SYSTEM_SERVICE)—Cannot specify both descriptor and name
ES$IPM	(type: SYSTEM_SERVICE)—Illegal parameter; the value of the data-length parameter exceeds the size of the ring buffer for a record-mode operation
ES$IST	(type: SYSTEM_SERVICE)—Illegal structure descriptor; no such ring buffer exists
ES$MDN	(type: SYSTEM_SERVICE)—Must specify descriptor or name

The request may return the following error, though not as a result of standard Pascal programming practice:

ES$IAD	(type: SYSTEM_SERVICE)—Invalid address; pointer to buffer or structure is odd or not in user address space

15.7 GET_ELEMENT_ANY

MACRO equivalent: GELA$

The GET_ELEMENT_ANY function implements a complex form of the GET_ELEMENT request; see the GET_ELEMENT and PUT_ELEMENT requests for a description of the basic get and put element operations on ring buffers. GET_ELEMENT_ANY performs the basic get element operation on the logical OR of several ring buffers, with an optional timeout feature. That is, GET_ELEMENT_ANY permits the calling process to test for and, if necessary, wait on an available data record in any one of a set of ring buffers. Up to four ring buffers can be specified in the request.

The function returns ordinal values from 0 to 5 to indicate the results of the operation (see Semantics).

If a complete record is immediately available in any of the specified ring buffers, the calling process gets the record and continues execution. Otherwise, the calling process blocks until one of the ring buffers can provide the requested number of bytes. More specifically, if each of the specified ring buffers is initially empty or contains less than a full record, the caller blocks on all the buffers. (Partial data transfers never occur, due to the mandatory record mode output access.) The process waits until the full request can be satisfied by any one of the ring buffers, at which point it is unblocked from all of them.

Optionally, the operation can be terminated due to the expiration of a time interval specified in the request.

Thus, GET_ELEMENT_ANY allows a process to get a specified number of bytes from any of up to four ring buffers, although it might be used primarily for its optional timeout capability.

If a zero time period (immediate timeout) is specified in the request, the GET_ELEMENT_ANY provides a complex form of the COND_GET_ELEMENT operation, which tests for available data but will not block the caller. See COND_GET_ELEMENT for a description of the basic conditional get operation.

Syntax

```
GET_ELEMENT_ANY (  ⟦ SDB4 := ring-buffer-descriptor-4 ⟧
                   ⟦ SDB3 := ring-buffer-descriptor-3 ⟧
                   ⟦ SDB2 := ring-buffer-descriptor-2 ⟧
                   SDB1 := ring-buffer-descriptor-1
                   LENGTH := data-length
                   DATA := input-buffer-id
                   ⟦ TIMEOUT := timeout-interval ⟧
                   ⟦ STATUS := status-record ⟧ )
```

ring-buffer-descriptor-4
ring-buffer-descriptor-3
ring-buffer-descriptor-2
ring-buffer-descriptor-1
The identifier of a variable of predefined type RING_BUFFER_DESC that contains a ring buffer's structure identifier.

Each ring buffer's output access mode must be record mode. You can specify up to four ring buffers. The order in which you specify multiple ring buffers determines the order in which they are initially tested for data bytes. That order can be critical under certain real-time conditions, as discussed under Semantics and Implementation Notes. Each descriptor must have been previously initialized by a CREATE_RING_BUFFER or an INIT_STRUCTURE_DESC request.

data-length

A constant or the identifier of a variable of type RING_BUFFER_DATA that specifies the number of bytes to read from the ring buffer.

input-buffer-id

The identifier of the variable (buffer) that is to receive the ring buffer data.

timeout-interval

The identifier of a variable of predefined type LONG_INTEGER that specifies the maximum time, in milliseconds, that the caller wishes to be blocked waiting for data. The value must be a positive integer from 0 to (2**31) –1. A value of 0 causes the request to time out immediately if none of the specified ring buffers has an entire record when GET_ELEMENT_ANY is called. That is, the calling process will never block if the specified time interval is 0. If you do not specify this parameter, the function assumes no timeout for the operation; the calling process may block indefinitely.

status-record

The identifier of a variable of predefined record type EXC_STATUS that may receive an exception type and code. If you specify this parameter, the exception status, either success or error, that results from issuing the request is reported in this variable. Otherwise, an error causes the corresponding exception to be reported. The format of the exception record is described in Section 11.1.2.

Restrictions

- Each ring buffer's output access mode must be record mode (see CREATE_RING_BUFFER).
- The number of bytes requested in the data-length parameter must not exceed the size of the ring buffer(s) specified in the CREATE_RING_BUFFER request.
- The timeout-interval value is limited to a 31-bit positive integer; that is, the sign bit of the high-order word must not be set. The maximum valid value, in milliseconds, permits a timeout period of just over 24.89 days (see SLEEP for more detail).
- If you wish to use fewer than four ring buffer descriptor parameters, you must assign them beginning with keyword SDB1. You may not assign keyword parameters with higher numbered suffixes unless all keywords with lower-numbered suffixes are assigned. For example, if the parameter sequence specifies keyword SDB3, the sequence must also include keywords SDB2 and SDB1.

Example

```
%INCLUDE 'COMPLX.PAS'

VAR
  Ring_1, Ring_2, Ring_3 : RING_BUFFER_DESC;
  Ch : CHAR;
  Which_one : COMPLEX_FUNC_VALUE;
  Timeout_val : LONG_INTEGER;

[PRIORITY(10), STACK_SIZE(100), NAME ('READER')] PROCESS Reader;
BEGIN

  (* Get a character from one of three ring buffers. *)
  Which_one := GET_ELEMENT_ANY
                (LENGTH := 1,
                 DATA := Ch,
                 SDB1 := Ring_1,
                 SDB2 := Ring_2,
                 SDB3 := Ring_3);

  (* Get a character from one of two ring buffers with timeout. *)
  Timeout_val := 1000;
  Which_one := GET_ELEMENT_ANY
                (LENGTH := 1,
                 DATA := Ch,
                 SDB1 := Ring_1,
                 SDB2 := Ring_2,
                 TIMEOUT := Timeout_val);

END; (* Process Reader *)
```

Semantics

For clarity, the following description ignores the unlikely case of multiple waiting processes for ring buffer output. That is, the description assumes that only one process is attempting to get data from a given ring buffer, although GET_ELEMENT_ANY allows for the possibility of multiple getters and guarantees sequential access, as does the basic GET_ELEMENT request.

The GET_ELEMENT_ANY function tests each of the ring buffers specified in the request for two conditions: at least data-length bytes available or fewer than data-length bytes available. The ring buffers are tested in the keyword order SDB1 to SDB4.

The function returns the following values:

Value	Meaning
0	Request timed out
1	Request satisfied by SDB1
2	Request satisfied by SDB2
3	Request satisfied by SDB3
4	Request satisfied by SDB4
5	Error condition

If any of the ring buffers contains at least data-length bytes at the time of the call, the function transfers data-length bytes from the first such ring buffer encountered and returns to the caller, with an integer value between 1 and 4 that indicates which ring buffer specified in the call satisfied the request.

If all the ring buffers contain less than data-length bytes and a zero timeout-interval was supplied in the call, the function returns immediately to the caller with a zero value, indicating a return due to timeout. The calling process thus never leaves the run state in the case of an immediate timeout.

If all the ring buffers contain less than data-length bytes and either no timeout-interval or a nonzero timeout-interval was supplied in the call, the function switches the calling process to the wait-active state. The process is blocked on each of the ring buffers specified in the request. The calling process remains blocked on all the ring buffers until at least data-length bytes of data (a full record) accumulates in any one of them. At that point, the function performs the requested data transfer, unblocks the caller from all the ring buffers, and switches the caller to the ready-active state and returns a nonzero ordinal value as described above.

In the case of process blocking described above, if a nonzero timeout-interval was supplied in the call, the calling process is also blocked on an internal timer queue, as well as on one or more ring buffers. If the specified timeout period expires at any point before the request can be satisfied, the caller is removed from all blocking structures and is switched to the ready-active state and returns a 0 value. Any partial record(s) accumulated at that point remain in the respective ring buffer(s).

If an error occurs, the function returns a 5.

This request is implemented through the GELA$ kernel primitive.

Error Returns

See Section 11.2 for general information about error returns. The following exception codes may be returned:

ES$IPM	(type: SYSTEM_SERVICE)—Illegal parameter; data-length value exceeds size of one of the ring buffers specified in the request, or timer value out of range
ES$IPR	(type: SYSTEM_SERVICE)—Illegal primitive; the output access mode of one of the ring buffers is not record mode
ES$IST	(type: SYSTEM_SERVICE)—Invalid structure descriptor; no such ring buffer exists

The request may return the following error, though not as a result of standard Pascal programming practice:

ES$IAD	(type: SYSTEM_SERVICE)—Invalid address; timer-value pointer is an odd address

Implementation Notes

Since the initial test of the ring buffers for a complete record is performed in determinate order, the order in which multiple buffers are associated with the parameters in the call can be critical under certain real-time conditions. For example, assume that the relative frequency of puts is high for one of several ring buffers and that the "fast" ring buffer is identified as first by being associated with keyword SDB1. In a series of calls to GET_ELEMENT_ANY, that ring buffer will tend to be serviced far more often than the others, and the "slower" ring buffers may seldom or never be tested and serviced. Optimally, then, the ring buffer with the highest expected signal rate should be assigned to the keyword that is tested last; the next highest as next to last, and so on, assuming that probable relative frequencies can be determined. Alternatively, the order in which the ring buffers are identified could be rotated in successive calls so at least N buffers are guaranteed to be tested in N calls to GET_ELEMENT_ANY.

As a contrary example, assume that the specified set of ring buffers represent device inputs (a common use) and that one of the devices has the highest priority in terms of its need to be serviced. The "service priority" might be independent of expected input rates, which if different could be reflected by differing ring buffer sizes, for example. In that case, the highest-priority buffer would be identified as first in the call to GET_ELEMENT_ANY, ensuring that that buffer is always tested on any call.

The correct or best-case strategy depends on application-specific factors, of course.

15.8 GET_VALUE

MACRO equivalent: GVAL$

The GET_VALUE procedure obtains the value and type code of a specified structure. The code identifies a structure as a binary, counting, or queue semaphore or as a ring buffer. The meaning of the structure's value depends on the structure type. For example, the value of a counting semaphore is the current signal count, whereas the value of a ring buffer is the current element count.

Note

The value of a structure may change immediately after it is inspected. Therefore, the information this request provides must be used cautiously, to prevent the introduction of race conditions.

Syntax

```
GET_VALUE (  VALUE := count
             TYP := structure-type
             { DESC := descriptor }
             { NAME := name       }
             [[ STATUS := status-record ]] )
```

count

The identifier of a variable of type INTEGER that will receive the structure's value.

structure-type

The identifier of a variable of type INTEGER that will receive the structure's type code.

descriptor

The identifier of a variable of predefined type RING_BUFFER_DESC that contains the semaphore's structure identifier. The variable must have been previously initialized by an appropriate CREATE type request or an INIT_STRUCTURE_DESC request.

name

A character-string constant or the identifier of a variable of predefined type NAME_STR that specifies the 6-character name of an existing semaphore (see Section 11.1.1.1).

status-record

The identifier of a variable of predefined record type EXC_STATUS that may receive an exception type and code. If you specify this parameter, the exception status, either success or error, that results from issuing the request is reported in this variable. Otherwise, an error causes the corresponding exception to be reported. The format of the exception record is described in Section 11.1.2.

Structure Type Identification Codes

The type codes and meaning of the values that the procedure can return are:

Structure	Type Code	Meaning of Value Parameter
Binary semaphore	0	The value of the gate variable (0 or 1)
Counting semaphore	1	The count of pending signals (0 or positive)
Queue semaphore	2	The count of pending signals (0 or positive)
Ring buffer	3	The count of data bytes in the ring buffer
PCB	4	No meaning
SRD	5	No meaning
Unformatted	7	No meaning

Example

```
VAR
  Sem_val, Sem_typ : INTEGER;
  Ring : RING_BUFFER_DESC;

[PRIORITY(10), STACK_SIZE(100)] PROCESS P1;
BEGIN

  (* Get the value of an unnamed ring buffer. *)
  GET_VALUE
    (VALUE := Sem_val, TYP := Sem_typ, DESC := Ring);
  (* Get the value of a named ring buffer. *)
  GET_VALUE
    (VALUE := Sem_val, TYP := Sem_typ, NAME := 'RING  ');

END; (* Process P1 *)
```

Semantics

The GET_VALUE procedure obtains the type code and structure value of the specified structure, stores that information in the variables specified in the call, and returns control to the caller.

This request is implemented through the GVAL$ kernel primitive.

Error Returns

See Section 11.2 for general information about error returns. The following exception codes may be returned:

ES$CDN	(type: SYSTEM_SERVICE)—Cannot specify both descriptor and name
ES$IPR	(type: SYSTEM_SERVICE)—Illegal primitive; the descriptor or name parameter is a logical name that does not translate to the name of a structure
ES$IST	(type: SYSTEM_SERVICE)—Illegal structure descriptor; no such structure exists
ES$MDN	(type: SYSTEM_SERVICE)—Must specify descriptor or name

15.9 INIT_STRUCTURE_DESC

MACRO equivalent: GVAL$

The INIT_STRUCTURE_DESC procedure copies identifying information about a specified ring buffer into a structure descriptor record. That record provides the kernel with a rapid-access path to the ring buffer referred to in the other ring buffer management requests.

You may also set up a structure descriptor record by using the CREATE_RING_BUFFER request when you create a ring buffer.

Syntax

```
INIT_STRUCTURE_DESC (  DESC := descriptor
                       NAME := name
                       ⟦ STATUS := status-record ⟧ )
```

descriptor

The identifier of a variable of predefined type RING_BUFFER_DESC that will receive the ring buffer's structure identifier.

name

A character-string constant or the identifier of a variable of predefined type NAME_STR that contains the 6-character name of an existing ring buffer (see Section 11.1.1.1).

status-record

The identifier of a variable of predefined record type EXC_STATUS that may receive an exception type and code. If you specify this parameter, the exception status, either success or error, that results from issuing the request is reported in this variable. Otherwise, an error causes the corresponding exception to be reported. The format of the exception record is described in Section 11.1.2.

Example

```
VAR
  Ring : RING_BUFFER_DESC;

[PRIORITY(10), STACK_SIZE(100), NAME ('P1    ')] PROCESS P1;
BEGIN

  (* Get the id of the ring buffer named 'RING  '. *)
  INIT_STRUCTURE_DESC (DESC := Ring, NAME := 'RING  ');

END; (* Process P1 *)
```

Semantics

The INIT_STRUCTURE_DESC procedure requests the kernel to copy the structure identifier, consisting of the index and serial number associated with the structure identified in the name parameter, into the structure descriptor record specified in the descriptor parameter.

This request is implemented through the GVAL$ kernel primitive.

Error Returns

See Section 11.2 for general information about error returns. The following exception code may be returned:

ES$IST	(type: SYSTEM_SERVICE)—Illegal structure descriptor; no such ring buffer exists

15.10 PUT_ELEMENT

MACRO equivalent: PELM$

The PUT_ELEMENT procedure copies a specified number of data bytes from the caller's storage area to a ring buffer. If the ring buffer has insufficient space to contain the number of bytes specified, the calling process blocks on the ring buffer until space becomes available.

In general, if two or more processes put data into the same ring buffer, the calling process will block if another process is waiting for its PUT_ELEMENT request to be satisfied. In that case, the calling process must wait its turn for access to the buffer, since sequential access to a ring buffer is ensured among multiple writers as well as among multiple readers. The process that blocks first is given active write access to the buffer and is never displaced by another, higher-priority process, regardless of the ordering attribute of the waiting-input-process list.

If the ring buffer has stream mode, the data transfer may occur in increments while the process is waiting. Essentially, stream-mode access permits a ring buffer to be smaller than the "records" that may be passed through it.

The COND_PUT_ELEMENT is the conditional, or nonblocking, form of the PUT_ELEMENT request.

The GET_ELEMENT and COND_GET_ELEMENT requests allow a process to extract bytes from a ring buffer, freeing the corresponding space.

Syntax

```
PUT_ELEMENT (  LENGTH := data-length
               DATA := data-id
               { DESC := ring-buffer-descriptor }
               { NAME := ring-buffer-name       }
               [[ STATUS := status-record ]] )
```

data-length

A constant or the identifier of a variable of type RING_BUFFER_DATA that specifies the number of bytes of data to copy into the ring buffer. This value is the length of the data identified by the data-id parameter.

data-id

The identifier of the variable (buffer) that contains data to be copied into the ring buffer.

ring-buffer-descriptor

The identifier of a variable of predefined type RING_BUFFER_DESC that contains the ring buffer's structure identifier. The variable must have been previously initialized by a CREATE_RING_BUFFER or an INIT_STRUCTURE_DESC request.

ring-buffer-name

A character-string constant or the identifier of a variable of predefined type NAME_STR that contains the 6-character name of an existing ring buffer (see Section 11.1.1.1).

status-record

The identifier of a variable of predefined record type EXC_STATUS that may receive an exception type and code. If you specify this parameter, the exception status, either success or error, that results from issuing the request is reported in this variable. Otherwise, an error causes the corresponding exception to be reported. The format of the exception record is described in Section 11.1.2.

Restriction

If the ring buffer's input access mode is record mode, the number of bytes specified by the data-length parameter must not exceed the size of the ring buffer (as specified in the CREATE_RING_BUFFER request); otherwise, the request will never be successful.

Example

```
VAR
  Ring_1 : RING_BUFFER_DESC;

[PRIORITY(10), STACK_SIZE(100), NAME ('WRITER')] PROCESS Writer;
BEGIN

  (* Put a character into an unnamed ring buffer. *)
  PUT_ELEMENT (LENGTH := 1,
               DATA := 'A',
               DESC := Ring_1);

  (* Put a character into a named ring buffer. *)
  PUT_ELEMENT (LENGTH := 1,
               DATA := 'A',
               NAME := 'RING1 ');

END; (* Process Writer *)
```

Semantics

If no other process is waiting to put data into the ring buffer, the PUT_ELEMENT procedure tests the ring buffer for the number of bytes of available space specified by the data-length parameter. If at least that amount of space is available, the procedure copies the data from the variable specified by the data-id parameter to the ring buffer and returns control to the caller.

If the ring buffer has insufficient space for the entire transfer and the input access mode is RECORD_MODE, the procedure blocks the calling process with active write access to the ring buffer and calls the scheduler. When enough additional space becomes available as a result of one or more subsequent GET_ELEMENT, COND_GET_ELEMENT, or GET_ELEMENT_ANY requests, the transfer is performed, and the waiting process is unblocked.

If the ring buffer has insufficient space for the entire transfer and the input access mode is stream mode, the procedure blocks the caller with active write access to the ring buffer, copies the bytes that can be accommodated, if any, and calls the scheduler. When enough additional space becomes available as a result of one or more subsequent GET_ELEMENT operations, the transfer is completed (possibly by a series of partial transfers) and the waiting process is unblocked.

If other processes are waiting to put data into the ring buffer at the time of the call, implying that some other process has active write access, the calling process is also blocked and is added to the ring buffer's list of waiting-to-put processes at a position below the head of the list. Processes are queued on the waiting process list in either FIFO or priority order, depending on the ring buffer definition (see the CREATE_RING_BUFFER function). (A process with active access is never displaced by another process, regardless of relative priorities.) The process waits its turn to gain active write access, at which point the process is treated as described above.

This request is implemented through the PELM$ kernel primitive.

Error Returns

See Section 11.2 for general information about error returns. The following exception codes may be returned:

ES$CDN	(type: SYSTEM_SERVICE)—Cannot specify both descriptor and name
ES$IPM	(type: SYSTEM_SERVICE)—Illegal parameter; the value of the data-length parameter exceeds the size of the ring buffer for a record-mode operation
ES$IST	(type: SYSTEM_SERVICE)—Illegal structure descriptor; no such ring buffer exists
ES$MDN	(type: SYSTEM_SERVICE)—Must specify descriptor or name

The request may return the following error, though not as a result of standard Pascal programming practice:

ES$IAD	(type: SYSTEM_SERVICE)—Invalid address; pointer to buffer or structure is odd or not in user address space

15.11 RESET_RING_BUFFER

MACRO equivalent: RBUF$

The RESET_RING_BUFFER procedure resets a ring buffer by emptying it of data bytes. This procedure allows a process to cancel an I/O sequence and to effectively empty the associated ring buffer without issuing multiple GET_ELEMENT requests.

This request is like a GET_ELEMENT request in that the calling process is treated as a "getting" process for synchronization. That is, if any other process is blocked on the ring buffer, waiting for a GET_ELEMENT request to be satisfied, the calling process is blocked and must wait its turn for read access to the buffer in the same manner as it would for a GET_ELEMENT request.

Note

> The RESET_RING_BUFFER request does not inhibit a concurrent attempt by another process to copy bytes into the buffer. Thus, in certain applications the ring buffer may not be empty by the time that control returns to the caller.

Syntax

```
RESET_RING_BUFFER ( { DESC := ring-buffer-descriptor }
                    { NAME := ring-buffer-name       }
                   [[ STATUS := status-record ]] )
```

ring-buffer-descriptor

The identifier of a variable of predefined type RING_BUFFER_DESC that contains the ring buffer's structure identifier. The variable must have been previously initialized by a CREATE_RING_BUFFER or an INIT_STRUCTURE_DESC request.

ring-buffer-name

A character-string constant or the identifier of a variable of predefined type NAME_STR that contains the 6-character name of an existing ring buffer (see Section 11.1.1.1).

status-record

The identifier of a variable of predefined record type EXC_STATUS that may receive an exception type and code. If you specify this parameter, the exception status, either success or error, that results from issuing the request is reported in this variable. Otherwise, an error causes the corresponding exception to be reported. The format of the exception record is described in Section 11.1.2.

Example

```
VAR
  Ring_1 : RING_BUFFER_DESC;

[PRIORITY(10), STACK_SIZE(100), NAME ('P1    ')] PROCESS P1;
BEGIN

  (* Reset an unnamed ring buffer. *)
  RESET_RING_BUFFER (DESC := Ring_1);

  (* Reset a named ring buffer. *)
  RESET_RING_BUFFER (NAME := 'RING1 ');

END; (* Process P1 *)
```

Semantics

If no other process is waiting to obtain bytes from the ring buffer, the RESET_RING_BUFFER procedure deletes any available bytes from the buffer and returns control to the caller.

If another process is waiting to obtain bytes from the ring buffer, the procedure places the calling process on the ring buffer's waiting-to-get process list at a position below the head of the list (as described for a GET_ELEMENT request) and calls the scheduler. When the blocked process gains read access to the ring buffer, the buffer is emptied, and the process is unblocked.

This request is implemented through the RBUF$ kernel primitive.

Error Returns

See Section 11.2 for general information about error returns. The following exception codes may be returned:

ES$CDN	(type: SYSTEM_SERVICE)—Cannot specify both descriptor and name
ES$IST	(type: SYSTEM_SERVICE)—Illegal structure descriptor; no such ring buffer exists
ES$MDN	(type: SYSTEM_SERVICE)—Must specify descriptor or name

Chapter 16
Interrupt Management Requests

This chapter describes the requests that let a process establish a connection to an interrupt vector so the process can respond to the occurrence of an interrupt. Those requests, implemented through the predeclared procedures listed in Table 16–1, are the Pascal language interface to the services provided by the kernel's interrupt management primitives. The requests let device handlers written in Pascal associate interrupts with their interrupt service routines (ISRs) and signal semaphores when interrupts occur.

Chapter 7 of the *MicroPower/Pascal Run-Time Services Manual* contains detailed information on MicroPower/Pascal device handling, focusing on ISRs and their association with device handlers. You should be thoroughly familiar with the contents of that chapter.

You must include in your program or module the definitions of the routines that implement the requests described in this chapter before using them. See Appendix I for more information.

Table 16–1 summarizes these requests.

Table 16–1: Interrupt Management Requests

Request	Operation
CONNECT_INTERRUPT	Associates an interrupt vector with an interrupt service routine to establish a process as a device handler
CONNECT_SEMAPHORE	Associates an interrupt vector with a semaphore so the semaphore is signaled each time an interrupt occurs
DISCONNECT_INTERRUPT	Breaks the connection between an interrupt vector and an interrupt service routine so interrupts from that vector are ignored
DISCONNECT_SEMAPHORE	Breaks the connection between an interrupt vector and the semaphore to which it was connected so interrupts from the vector are ignored

The CONNECT_SEMAPHORE/DISCONNECT_SEMAPHORE combination can be used where interrupts occur at a relatively low rate for the amount of data being transmitted, as with direct memory access (DMA) devices that generate an interrupt when a block of data has been transferred or with slow devices, such as terminals. Those procedures let you perform device servicing at process level by using a semaphore to communicate the occurrence of an interrupt.

The CONNECT_INTERRUPT/DISCONNECT_INTERRUPT combination is used where interrupts occur at a relatively high rate for the amount of data being obtained, as with fast data rate serial devices like DECtape II, which generate an interrupt for each character. Using these procedures, you must create your own interrupt service routine in MACRO-11 assembly language to perform device servicing at interrupt level rather than at process level. Doing so eliminates the overhead cost of process-context switching for each interrupt.

16.1 CONNECT_INTERRUPT

MACRO equivalent: CINT$

The CONNECT_INTERRUPT procedure associates an interrupt vector with an interrupt service routine (ISR) specified in the call. The procedure allows a process to establish itself as a device handler and, unlike the CONNECT_SEMAPHORE procedure, allows the process to define the ISR code segment. The ISR must be coded in MACRO–11 assembly language. Chapter 7 of the *MicroPower/Pascal Run-Time Services Manual* describes the coding requirements for ISRs.

This request is normally used only by a process declared with the DEV_ACCESS or the DRIVER attribute, since access to the I/O page is generally required for device handling (see Section 10.1.2).

Syntax

```
CONNECT_INTERRUPT ( PIC := pic-indicator
                    ISR := isr-entry-point
                    IMPURE := impure-area-pointer
                    [[ VALUE := r4-value ]]
                    PS := program-status-word
                    VECTOR := interrupt-vector-address )
```

pic-indicator

The identifier of a variable of type BOOLEAN; when TRUE, indicates that the ISR is written in position-independent code (PIC).

isr-entry-point

The name of an external procedure that names the MACRO–11 global identifier for the ISR.

impure-area-pointer

The identifier of a variable of predefined type UNIVERSAL that will be passed to the ISR in processor register R3.

r4-value

A constant or the identifier of a variable of type UNSIGNED that specifies an arbitrary value to be passed to the ISR in processor register R4 on interrupt dispatch. The default value is 0. (Typical uses of this argument are to pass a device address, table index, or other means of identifying the vector causing the interrupt, in the case of an ISR connected to several vectors.)

program-status-word

A constant or the identifier of a variable of predefined type PRIORITY_RANGE that specifies the content of the PSW desired on dispatch to the ISR. This argument has the form [PS:=]word-value, with the effective PSW value in the low byte. The primary effect of this parameter is to set the processor priority level at which the ISR is to execute when entered. If priority-level 7 is requested (that is, PS = 340(octal)), a special form of ISR dispatching is implied (see Chapter 7 of the *MicroPower/Pascal Run-Time Services Manual*). You can also set the CC bits with this argument, but not the T-bit.

interrupt-vector-address

A constant or the identifier of a variable of predefined type UNSIGNED that specifies the address of the interrupt vector to be connected to the ISR.

Restriction

A module that contains a CONNECT_INTERRUPT request should not be added to a supervisor-mode shared library.

Example

```
TYPE
  Unit_range = 0..15;

VAR
  Unit : Unit_range;
  Impure_ptr : UNIVERSAL;
[EXTERNAL(consol)] PROCEDURE Console; EXTERNAL;

[INITIALIZE] PROCEDURE Init;
BEGIN

  (* Set up to catch interrupts through vector 60(octal). *)
  CONNECT_INTERRUPT
    (PIC := TRUE,  (* The ISR is position independent. *)
    ISR := CONSOLE,  (* External Interrupt Service Routine *)
    IMPURE := Impure_ptr, (* Pointer to impure area *)
    VALUE := unit,  (* Contents of R4 at entry to ISR *)
    PS := %O'200',  (* Priority 4 *)
    VECTOR := %O'60');

END; (* Procedure Init *)
```

Semantics

The CONNECT_INTERRUPT procedure sets up the interrupt dispatch block (IDB) associated with the specified vector to cause interrupts to be dispatched to the specified ISR entry point. The procedure also identifies the caller as the process owning the connected vector (see the DISCONNECT_INTERRUPT procedure).

This request is implemented through the CINT$ kernel primitive.

Note

Chapter 7 of the *MicroPower/Pascal Run-Time Services Manual* contains information closely related to the use of CONNECT_INTERRUPT and the coding of ISRs. The chapter describes interrupt dispatching, which is affected by certain CONNECT_INTERRUPT parameters (especially the PSW value) and the kernel/ISR interface.

Error Returns

See Section 11.2 for general information about error returns. The following exception codes may be returned:

ES$AOV	(type: SYSTEM_SERVICE)—Already owned vector; the specified vector is already connected
ES$IAD	(type: SYSTEM_SERVICE)—Invalid address; invalid ISR mapping (mapped systems only)
ES$IVC	(type: SYSTEM_SERVICE)—Illegal vector; the specified vector address is less than 60(octal) or beyond the valid range of vectors established at build time (PROCESSOR macro)
ES$NID	(type: SYSTEM_SERVICE)—No IDB established for vector; the vector address was not specified in the DEVICES macro of the system configuration file

16.2 CONNECT_SEMAPHORE

MACRO equivalent: CINT$

The CONNECT_SEMAPHORE procedure associates an interrupt vector with a binary or a counting semaphore so the semaphore is signaled each time an interrupt occurs through that vector. A process can thus establish itself as a device handler.

This request is normally used only by a process declared with either the DEV_ACCESS or the DRIVER attribute, since access to the I/O page is generally required for device handling (see Section 10.1.2).

Syntax

```
CONNECT_SEMAPHORE ( VECTOR := interrupt-vector-address
                    PS := program-status-word
                    DESC := sem-descriptor
                    [[ NAME := sem-name ]]
                    [[ STATUS := status-record ]] )
```

interrupt-vector-address

A constant or the identifier of a variable of type UNSIGNED that specifies the address of the interrupt vector to be connected to the specified semaphore.

program-status-word

A constant or the identifier of a variable of predefined type PRIORITY_RANGE that specifies the content of the PSW desired on dispatch to the ISR. This argument has the form [PS:=]word-value, with the effective PSW value in the low byte. The primary effect of this parameter is to set the processor priority level at which the ISR is to execute when entered.

sem-descriptor

The identifier of a variable of predefined type SEMAPHORE_DESC that contains the semaphore structure identifier. The variable must have been previously initialized by a CREATE_BINARY_SEMAPHORE, a CREATE_COUNTING_SEMAPHORE, or an INIT_STRUCTURE_DESC request.

sem-name

A character-string constant or the identifier of a variable of predefined type NAME_STR that specifies the 6-character name of a binary or a counting semaphore.

status-record

The identifier of a variable of predefined record type EXC_STATUS that may receive an exception type and code. If you specify this parameter, the exception status, either success or error, that results from issuing the request is reported in this variable. Otherwise, an error causes the corresponding exception to be reported. The format of the exception record is described in Section 11.1.2.

Restrictions

The specified semaphore must not be a queue semaphore.

Examples

1. The following program segment shows how to use CONNECT_SEMAPHORE to connect to the clock interrupt vector.

```
VAR
Isr_sem : SEMAPHORE_DESC;

PROCEDURE Tick;
  BEGIN
    WHILE TRUE DO
      BEGIN
        WAIT (DESC := Isr_sem);
        WRITELN ('Tick');
      END;
  END; (* Procedure Tick *)

[INITIALIZE] PROCEDURE Init;
  BEGIN

    (* Set up to catch interrupts from the clock. *)
    CONNECT_SEMAPHORE (VECTOR := %O'100',  (* The clock *)
                       PS := %O'300',      (* Priority 6 *)
                       DESC := Isr_sem);

  END; (* Procedure Init *)
```

2. This program is a device driver for a DL serial line.

```
[DATA_SPACE(3000),PRIORITY(200)] PROGRAM xl;

  TYPE
    rcver = PACKED RECORD
      interrupt_enable  : [POS(6)] BOOLEAN;
                  done  : [POS(7)] BOOLEAN;
                  datai : [POS(16)] CHAR;
                  error : [POS(31)] BOOLEAN;
                END;
    xmitter = PACKED RECORD
      interrupt_enable  : [POS(6)] BOOLEAN;
                  ready : [POS(7)] BOOLEAN;
                  datao : [POS(16)] CHAR;
                  error : [POS(31)] BOOLEAN;
                END;

  VAR
    receiver : [AT(%O'176500'), VOLATILE] rcver;
    transmitter : [AT(%O'176504'), VOLATILE] xmitter;
    in_rb,out_rb : RING_BUFFER_DESC;
    in_interrupt, out_interrupt : SEMAPHORE_DESC;
    xl_error : BOOLEAN;
    i : INTEGER;
```

```
[INITIALIZE] PROCEDURE a;
  BEGIN
    IF (CREATE_RING_BUFFER (DESC:=in_rb,
                            NAME:='XLIO  ',
                            SIZE:=80)
      AND CREATE_RING_BUFFER (DESC:=out_rb,
                              NAME:='XLOO  ', SIZE:=80)
      AND CREATE_BINARY_SEMAPHORE (DESC:=in_interrupt,
                                   NAME:='INTRPT')
      AND CREATE_BINARY_SEMAPHORE (DESC:=out_interrupt,
                                   NAME:='OUTRPT'))
    THEN  xl_error := TRUE
    ELSE  xl_error := FALSE;
      CONNECT_SEMAPHORE (VECTOR:=%O'300',
                         PS:=0,
                         DESC:=in_interrupt);
      CONNECT_SEMAPHORE(VECTOR:=%O'304',
                        PS:=0,
                        DESC:=out_interrupt);
  END;

[TERMINATE] PROCEDURE b;
  BEGIN
    DISCONNECT_SEMAPHORE(VECTOR:=%O'300');
    DISCONNECT_SEMAPHORE(VECTOR:=%O'304');
    DISCONNECT_SEMAPHORE(VECTOR:=%O'100');
    CLOSE(INPUT);
    CLOSE(OUTPUT);
    DESTROY(NAME:='XLIO  ');
    DESTROY(NAME:='XLOO  ');
    DESTROY(NAME:='OUTRPT');
    DESTROY(NAME:='INTRPT');
  END;

[PRIORITY(210)] PROCESS rcv;
  VAR c : CHAR;
  BEGIN
    WITH receiver DO
      BEGIN
        interrupt_enable := TRUE;
        WHILE TRUE DO
          BEGIN
            WAIT (in_interrupt);
            c := datai;
            IF NOT error
            THEN  PUT_ELEMENT (DESC:=in_rb,
                               DATA:=c,
                               LENGTH:=1)
            ELSE  xl_error := TRUE;(* record that
                                      error occurred *)
          END;
      END;
  END;
```

```
[PRIORITY(209)] PROCESS xmt;
  VAR c : CHAR;
  BEGIN
    WITH transmitter DO
      BEGIN
        interrupt_enable:=true;
        WHILE TRUE DO
          BEGIN
            WAIT(DESC:=out_interrupt);
            GET_ELEMENT (DESC:=out_DATA:=c, length:=1);
            datao := c;
          END;
      END;
  END;

BEGIN        (* main program *)
  OPEN(INPUT,'XLIO:');
  OPEN(OUTPUT,'XLOO:');
  RESET(INPUT);
  REWRITE(OUTPUT);
  rcv(NAME:='rcv   ');
  xmt(NAME:='xmt   ');
  WHILE TRUE DO
    BEGIN
      WRITE('Enter number : ');
      READLN (i);
      WRITELN ('The number you typed was ', i:1);
    END; {WHILE TRUE DO ...}
END.
```

Semantics

The CONNECT_SEMAPHORE procedure sets up the IDB associated with the specified vector to cause interrupts to be dispatched to a predefined ISR. This ISR signals the semaphore each time an interrupt for that vector occurs. The procedure also identifies the caller as the process owning the connected vector (see the DISCONNECT_SEMAPHORE procedure).

This request is implemented through the CINT$ kernel primitive.

Error Returns

See Section 11.2 for general information about error returns. The following exception codes may be returned:

ES$AOV	(type: SYSTEM_SERVICE)—Already owned vector; the specified vector is already connected
ES$IAD	(type: SYSTEM_SERVICE)—Invalid address; invalid ISR mapping (mapped memory environments only)
ES$IVC	(type: SYSTEM_SERVICE)—Illegal vector; the specified vector address is less than 60(octal) or beyond the valid range of vectors established at build time (PROCESSOR macro)
ES$NID	(type: SYSTEM_SERVICE)—No IDB established for vector; the vector address was not specified in the DEVICES macro of the system configuration file

16.3 DISCONNECT_INTERRUPT

MACRO equivalent: DINT$

The DISCONNECT_INTERRUPT procedure breaks the connection between an interrupt vector and the ISR to which the procedure is connected. Further interrupts through that vector are ignored.

This procedure can be used only by the current owner of the vector, such as a termination procedure for a device driver process. (DISCONNECT_INTERRUPT will not ordinarily be used in a dedicated system environment but is supplied for functional completeness.)

Syntax

DISCONNECT_INTERRUPT (VECTOR := interrupt-vector-address)

interrupt-vector-address

The identifier of a variable of type UNSIGNED that specifies the address of the interrupt vector to be disconnected from the ISR.

Restrictions

- The specified vector, if connected, must have been connected by the calling process.
- A module that contains a CONNECT_INTERRUPT request should not be added to a supervisor-mode shared library.

Example

```
[TERMINATE] PROCEDURE Term;
BEGIN

  (* Stop fielding interrupts through vector 60(octal). *)
  DISCONNECT_INTERRUPT (VECTOR := %O'60');

END; (* Procedure Term *)
```

Semantics

The DISCONNECT_INTERRUPT procedure reinitializes the IDB associated with the specified vector to point to the kernel's null (do nothing) ISR. The null ISR dismisses any interrupts from unconnected vectors, after incrementing an unsolicited-interrupt counter.

If the specified vector is not connected at the time of the call, the procedure returns an illegal vector (ES$IVC) error.

This request is implemented through the DINT$ kernel primitive.

Error Returns

See Section 11.2 for general information about error returns. The following exception code may be returned:

ES$IVC	(type: SYSTEM_SERVICE)—Illegal vector; the specified vector address is less than 60(octal), or beyond the valid range of vectors established at build time (PROCESSOR macro), or is not connected or not owned by the calling process

16.4 DISCONNECT_SEMAPHORE

MACRO equivalent: DINT$

The DISCONNECT_SEMAPHORE procedure breaks the connection between an interrupt vector and the semaphore to which the procedure is connected. Further interrupts through that vector are ignored.

This procedure can be used only by the current "owner" of the vector, such as a termination procedure for a device driver process. (DISCONNECT_SEMAPHORE will not ordinarily be used in a dedicated system environment but is supplied for functional completeness.)

Syntax

```
DISCONNECT_SEMAPHORE (  VECTOR := interrupt-vector-address
                        ⟦ STATUS := status-record ⟧ )
```

interrupt-vector-address

A constant or the identifier of a variable of type UNSIGNED that specifies the address of the interrupt vector to be disconnected from the specified semaphore.

status-record

The identifier of a variable of predefined record type EXC_STATUS that may receive an exception type and code. If you specify this parameter, the exception status, either success or error, that results from issuing the request is reported in this variable. Otherwise, an error causes the corresponding exception to be reported. The format of the exception record is described in Section 11.1.2.

Example

```
[TERMINATE] PROCEDURE Term;
BEGIN

  (* Stop fielding interrupts from the clock. *)
  DISCONNECT_SEMAPHORE (VECTOR := %O'100');

END; (* Procedure Term *)
```

Semantics

The DISCONNECT_SEMAPHORE procedure reinitializes the IDB associated with the specified vector to point to the kernel's null (do nothing) ISR. The null ISR dismisses any interrupts from unconnected vectors, after incrementing an unsolicited-interrupt counter.

If the specified vector is not connected at the time of the call, the procedure returns control immediately to the caller, and no operation is performed.

This request is implemented through the DINT$ kernel primitive.

Error Returns

See Section 11.2 for general information about error returns. The following exception code may be returned:

ES$IVC	(type: SYSTEM_SERVICE)—Illegal vector; the specified vector address is less than 60(octal) or beyond the valid range of vectors established at build time (PROCESSOR macro)

Chapter 17
Exception Management Requests

This chapter describes the requests that let a process or a procedure manage the exceptions that may occur during application program execution. Those requests are the Pascal language interface to the services provided by the kernel's exception-handling primitives. Those procedures direct the dispatching and reporting of hardware and software exceptions to appropriate exception routines that you create.

The definitions of the routines that implement the requests described in this chapter must be included in your program or module before using them. See Appendix I for more information.

Table 17–1 summarizes the functions of the predeclared exception management requests.

Table 17–1: Exception Management Requests

Request	Operation
CONNECT_EXCEPTION	Establishes a process as the exception handler for a group of processes and for a specified exception type
DISCONNECT_EXCEPTION	Reverses the operation performed by the CONNECT_EXCEPTION procedure
ESTABLISH	Establishes a procedure as the exception handler for exceptions caused by the invoking process
RELEASE_EXCEPTION	Releases a process from the exception-wait state. The process may be passed to an exception-handling procedure for further processing, reinstated for normal operation, or aborted
REPORT	Allows a process to declare a software exception or to simulate a hardware exception or processor trap
REVERT	Reverses the operation performed by the ESTABLISH procedure
WAIT_EXCEPTION	Allows an exception-handling process to be notified that an exception occurred

An exception is a significant event associated with a processor trap (usually representing a hardware-detected error), a software-detected error, or other special condition. The condition indicated by an exception, such as a memory fault or an unavailable resource, generally casts doubt on the ability of the running process to continue normal execution. Exceptions, therefore, are essentially a kind of interrupt that, unlike I/O interrupts, causes a change in the flow of control within the running process. The kind of change depends on the kind of exception processing, if any, provided for the particular process and for the exception condition in question. The process may be switched to the exception-wait state, to await "attention" by a separate exception-handling process, control may be redirected to the process's own exception-service routine or procedure, or the process may be aborted—forced to an abnormal termination into the inactive state. (Any "unhandled" exception is fatal, causing the process to abort.)

Although 16 types of exceptions are defined for MicroPower/Pascal and many possible exception conditions exist within a given exception type, all exceptions can be loosely grouped into two categories: hardware exceptions and software exceptions. (Software exceptions constitute by far the larger category.) The characteristics of each category are as follows:

- Hardware exceptions result from processor traps that cause an exception to be raised directly and unconditionally by the kernel. (All traps except IOT, debugger-set breakpoint, and power-fail/restart, cause an exception.) Hardware exceptions represent either a hardware-detected error resulting from an instruction failure (an implicit error trap) or a special condition signaled by the intentional execution of a trap instruction in user code (an explicit "service trap"). Hardware-detected error conditions include bus timeouts, illegal or nonexistent addresses, illegal or reserved instructions, memory-parity errors, and memory-protection faults.

  The service traps are caused by EMT, TRP, and—possibly—BPT instructions. The BPT instruction, however, is used by the PASDBG symbolic debugger for setting dynamic breakpoints and thus is not recommended for use in source code. (The remaining trap instruction, IOT, implements normal entry to the kernel for primitive services and cannot be used for any other purpose.) Although trap instructions cannot be coded in a Pascal program, they can be simulated if necessary with the REPORT procedure.

- Software exceptions represent an error or other special condition that is detected by software and is conditionally raised as an exception via the Pascal REPORT procedure. In most cases, the condition is detected by a system software component, such as a kernel primitive or a system service process, which, in the case of some real-time and I/O requests, reports the condition back to the requesting process through the optional STATUS parameter, leaving the exception reporting to the discretion of the user process. The STATUS parameter (Section 11.1.2) is provided with those requests where a potential recovery is possible. The STATUS parameter suppresses the otherwise implicit, automatic reporting of SYSTEM_SERVICE and RESOURCE type exceptions by the Pascal OTS routines that support primitive service calls.

Many other software exceptions (mostly relating to I/O and arithmetic functions) are unconditionally reported as such by the generated code.

MicroPower/Pascal software allows you to create exception management routines at both the process and procedure levels.

An exception-handling process is one that has identified itself to the kernel by issuing a CONNECT_EXCEPTION request. This process manages exceptions for a group of processes specified by the GROUP parameter of the process declaration.

An exception-handling procedure is one that has been identified to the kernel in an ESTABLISH request issued by a process. An exception procedure manages exceptions for its parent process when an exception process either does not exist or dispatched the exception to it.

The hierarchical relationship of exception processes to exception procedures permits the implementation of a system-wide exception-handling policy. Chapter 6 of the *MicroPower/Pascal Run-Time Services Manual* provides detailed information about the kernel's exception management strategy.

17.1 Exception Types and Codes

Exceptions listed in Table 17–2 are categorized by type and code. The exception types identify the major exception categories and are members of the set type EXC_SET. The codes identify specific exceptions. The asterisks (*) in the table identify unconditional, kernel-raised exceptions that apply equally to Pascal and MACRO–11 processes.

Note

Do not confuse an exception type with an exception group, which is an attribute of a process. The exception group is a value that is specified when a process is created or declared. The group parameter (GROUP attribute) declares a process to be a member of a group for exception-handling purposes. (An exception group is a set of processes grouped because of common exception-handling requirements.) The group is then used to associate one or more exception handlers with a particular exception group.

The MicroPower/Pascal messages manual applicable to your host environment describes each exception in greater detail along with suggested recovery procedures.

Table 17–2: Exception Types and Codes

Type	Code	Description
Not Applicable		
	ES$NOR	Normal or successful completion of an operation. It is returned by the Pascal OTS only when you select the optional STATUS parameter provided with many of the requests.
MEMORY_FAULT *		
	ES$BUS *	Bus error: illegal address, timeout; trap to vector 4
	ES$MEM	Unspecified memory fault (subcode = 0); should never be encountered

Table 17–2 (Cont.): Exception Types and Codes

Type	Code	Description
	ES$MPT *	Memory-parity error, where applicable; trap to vector 114
	ES$MMU *	Memory protection error, mapped targets only; trap to vector 250
	ES$VEC *	Vector fetch error, FALCONs only; trap to vector 0
ILLEGAL_OPERATION *		
	ES$FOP *	FP–11 floating-point opcode error; trap to vector 244
	ES$ILL *	Illegal or reserved instruction; trap to vector 10
	ES$IOP	Unspecified illegal operation (subcode = 0); should never be encountered
EMULATOR_TRAP *		EMT instruction executed; trap to vector 30
	ES$EMT *	EMT instruction with a zero operand (subcode = 0)
	ES$xxx *	User-defined EMT exception codes, with subcode value from 1 to 255 matching EMT instruction operand
TRAP *		TRAP instruction executed, trap to vector 34
	ES$TRP *	TRAP instruction with a zero operand (subcode = 0)
	ES$xxx *	User-defined TRAP exception codes, with subcode value from 1 to 255 matching TRAP instruction operand
BREAKPOINT_TRAP *		These errors are not generated by PASDBG.
	ES$BPT *	User-coded BPT instruction executed; trap to vector 14
HARD_IO		Hard I/O errors returned a driver or communications process; corresponding exceptions are raised by the Pascal OTS
	ES$ABT	I/O request canceled by user or aborted by remote node
	ES$ATN	Device attention required
	ES$BOT	BOT (beginning of tape) encountered
	ES$CTL	Controller error
	ES$DAL	Device already allocated
	ES$DRV	Drive error
	ES$EVL	End of volume
	ES$FOR	Format error
	ES$FRM	Framing error
	ES$HIO	Unspecified hard I/O error (subcode = 0); should never be encountered

Table 17-2 (Cont.): Exception Types and Codes

Type	Code	Description
	ES$IBN	Invalid block number
	ES$IDA	Invalid device address
	ES$IVD	Invalid data
	ES$IVM	Invalid mode
	ES$IVP	Invalid parameter
	ES$NXM	Nonexistent or read-only memory
	ES$NXU	Nonexistent unit
	ES$OFL	Device off line or not mounted
	ES$OVF	Data overflow
	ES$OVR	Device overrun
	ES$PAR	Parity error
	ES$PNA	Packet not available to support request
	ES$PWR	Device power failure
	ES$SPD	I/O processing stopped
	ES$TIM	Device timeout
	ES$UNS	Unsafe volume
	ES$WLK	Write-locked unit
SOFT_IO		Soft I/O errors or special conditions, primarily returned by a driver, ACP, or communications process; the corresponding exceptions are raised by the Pascal OTS if the condition is unexpected. (Certain errors are detected as well as reported by the Pascal OTS.)
	ES$ABO	I/O aborted
	ES$BIV	Illegal Boolean value
	ES$DAS	Direct access requested on sequential file
	ES$DCF	Device full
	ES$DIO	Directory I/O error
	ES$DNU	Destination node is unreachable
	ES$DRF	Directory full
	ES$DVF	Attempt to signal device driver or ACP failed (detected by the OTS, ACP, or a communications process)
	ES$EOF	End of file encountered; not normally an error (detected by the ACP)

Table 17-2 (Cont.): Exception Types and Codes

Type	Code	Description
	ES$FAO	File already open
	ES$FIV	Illegal floating-point value
	ES$FNF	File not found
	ES$FNO	File not open
	ES$FNR	File not reset
	ES$FNW	File not rewritten
	ES$FRO	File is read-only: invalid write to OLD disk file
	ES$FVC	File-variable contention error
	ES$ICD	Invalid driver configuration data
	ES$IDR	Invalid directory format
	ES$IDS	Illegal device specification
	ES$IFN	Illegal function
	ES$IFS	Illegal file specification
	ES$IFW	Illegal field width
	ES$IIV	Illegal integer value
	ES$ILV	Illegal long integer value
	ES$INS	Invalid network specification
	ES$IRS	Illegal rename specification
	ES$IUP	Illegal use of UPDATE parameter
	ES$IVL	Invalid length specified
	ES$LRJ	Link rejected by remote task
	ES$NFS	Device not file structured
	ES$NIP	No I/O in progress
	ES$NRF	No reference data present
	ES$PAL	Path to remote task has been lost
	ES$PRO	File protection error
	ES$REF	Attempted read past EOF
	ES$RSZ	Record size of 0 specified
	ES$SIO	Unspecified soft I/O error (subcode = 0); should never be encountered

Table 17-2 (Cont.): Exception Types and Codes

Type	Code	Description
	ES$TNF	Task not found
	ES$UFN	Unsupported function
	ES$UIV	Illegal unsigned value
	ES$WEF	Attempted write past EOF
NUMERIC *		Numeric errors reported either by the kernel (floating-point traps) or by Pascal runtime checks
	ES$CON *	Floating-point conversion error (FP–11 only)
	ES$FDZ *	Floating-point divide by 0 (FP–11 or FIS)
	ES$FOV *	Floating-point overflow (FP–11 or FIS)
	ES$FUN *	Floating-point underflow (FP–11 or FIS)
	ES$IDZ	Integer divide by 0 (Pascal MATHCHECK option)
	ES$INM	Modulus of negative integer (Pascal MOD function)
	ES$IOV	Integer overflow (Pascal MATHCHECK option)
	ES$LDZ	Long integer divide by 0 (Pascal OTS)
	ES$LIC	Long integer to integer conversion error (Pascal OTS)
	ES$LNM	Modulus of negative long integer (Pascal OTS)
	ES$LNP	Log of nonpositive value (Pascal LN function)
	ES$LOV	Long integer overflow (Pascal OTS)
	ES$LUC	Long integer to unsigned conversion error
	ES$NUM	Unspecified numeric error (subcode = 0); should never be encountered
	ES$SRN	Square root of negative value (Pascal SQRT function)
	ES$UDV *	Undefined floating-point variable (FP–11 only)
	ES$UDZ	Unsigned divide by 0 (Pascal EIS/FIS/FPP OTS only)
	ES$UOV	Unsigned overflow (Pascal EIS/FIS/FPP OTS only)
RESOURCE		Resource errors, either returned by a primitive or system process and optionally reported by the Pascal OTS, or detected and reported only by the Pascal OTS
	ES$DDP	DISPOSE of already disposed pointer
	ES$LNR	Local node has no room for logical link
	ES$NFA	No free APR for window mapping

Table 17-2 (Cont.): Exception Types and Codes

Type	Code	Description
	ES$NFR	No free RAM
	ES$NLZ	NEW request of length 0
	ES$NMB	Insufficient data space for I/O buffer
	ES$NMC	Insufficient space for operation in RTACP pool
	ES$NMF	Insufficient data space for file variable
	ES$NMK	Insufficient pool space for kernel structure
	ES$NMP	Insufficient data space for user structure
	ES$NMS	Insufficient data space for stack
	ES$NNS	No network service process installed
	ES$RNR	Remote node has no room for logical link
	ES$RSC	Unspecified resource error (subcode = 0); should never be encountered
RANGE *		Range errors, detected and reported by Pascal runtime checks only, except as noted
	ES$ASO	Array subscript out of bounds (Pascal INDEXCHECK option)
	ES$CSO	Case selector out of range (Pascal RANGECHECK option)
	ES$NIL	Reference of a NIL pointer (Pascal POINTERCHECK option)
	ES$PCC	Program consistency check; should not occur
	ES$RAN	Unspecified range error (subcode = 0); should never be encountered
	ES$SEO	Set element out of range (Pascal RANGECHECK option)
	ES$STO *	Stack overflow (detected either by kernel or Pascal STACKCHECK option)
	ES$STU *	Stack underflow (detected either by kernel or Pascal STACKCHECK option)
	ES$VSE	Variable subrange exceeded (Pascal RANGECHECK option)
EXECUTION *		Execution error, pertaining to a FALCON or FALCON-PLUS target configuration option only
	ES$BRK *	FALCON break trap, if configured; trap to vector 140
	ES$EXC	Unspecified execution error (subcode = 0); should never be encountered
SYSTEM_SERVICE		System service errors returned mostly by primitive operations and optionally reported by the Pascal OTS; a few are detected and reported by the Pascal OTS only.

Table 17-2 (Cont.): Exception Types and Codes

Type	Code	Description
	ES$AOV	Already owned vector, cannot connect
	ES$CDN	Cannot specify both descriptor and name
	ES$EPN	Exception procedure not defined (Pascal REVERT request)
	ES$IAD	Invalid address: odd or not in user space
	ES$IPM	Illegal parameter
	ES$IPR	Illegal primitive
	ES$IST	Invalid structure descriptor
	ES$IVC	Illegal vector address
	ES$MDN	Must specify descriptor or name
	ES$NID	No interrupt dispatch block configured for vector
	ES$RDE	Reply descriptor expected by RECEIVE_ACK
	ES$SIU	Structure is in use
	ES$SNI	Structure name already in use
	ES$SVC	Unspecified system service exception (subcode = 0); should never be encountered
RESERVED_1		(Reserved by DIGITAL for future use)
RESERVED_2		(Reserved by DIGITAL for future use)
USER_1		Type reserved for user-defined, user-reported exceptions
	ES$US1	Nonspecific exception of type USER_1 (subcode = 0)
	ES$xxx	User-definable USER_1 type exception code, with subcode value from 1 to 2047 (3777 octal)
USER_2		Type reserved for user-defined, user-reported exceptions
	ES$US2	Nonspecific exception of type USER_2 (subcode = 0)
	ES$xxx	User-definable USER_2 type exception code, with subcode value from 1 to 2047 (3777 octal)

17.2 Format for Exception-Handling Procedure Declaration

You must use the following format to make sure that the procedure correctly receives the data the kernel passes to the procedure. Upon invocation by the kernel, an exception-handling procedure receives actual parameter values for the exception type and code, a pointer to the first word of a variable-length data item, and the length, in bytes, of that data item.

Syntax

```
PROCEDURE procedure-identifier ( exception-type : EXC_SET ;
                                 exception-code : EXC_CODES ;
                                 info-data-length : UNSIGNED ;
                                 VAR info-data-pointer : [READONLY] UNIVERSAL );
```

procedure-identifier

The identifier of the exception procedure.

exception-type

A parameter of predefined type EXC_SET that passes the exception type to the procedure (see Table 17–2).

exception-code

A parameter of type EXC_CODES that passes the exception code to the procedure.

info-data-length

A parameter of type UNSIGNED that passes to the procedure the number of bytes of information pointed to by the info-data parameter. A value of 0 indicates that no data is being passed; the pointer value in the info-data-pointer parameter is meaningless.

info-data-pointer

A VAR parameter of type [READONLY]UNIVERSAL that will receive a pointer to the first word of the optional argument list portion of the faulting process's exception stack frame. Both kernel-raised exceptions and the REPORT exception procedure provide information through this parameter. Chapter 6 of the *MicroPower/Pascal Run-Time Services Manual* provides the general format of the exception stack frame and the exception-specific argument lists supplied by various hardware exceptions.

17.3 CONNECT_EXCEPTION

MACRO equivalent: CCND$

The CONNECT_EXCEPTION procedure declares a process to be the exception handler for a group of processes and for a specified type exception. CONNECT_EXCEPTION establishes an existing queue semaphore, supplied by the caller, as the exception queue through which the specified exceptions will be signaled by the kernel. A process can thus be activated by and manage a specific type of exception (or possibly several types) when caused by any one of a group of processes having the same exception-group number.

A handler can use a single call to CONNECT_EXCEPTION to specify several exception types for one process group. Conversely, a handler must use a separate call to CONNECT_EXCEPTION when the same exception type applies to several process groups.

The queue element passed by the kernel by means of the exception queue semaphore to the handler process is the PCB of the process that caused the exception. The handler must be a process of high priority with the PRIVILEGED or DRIVER attribute to access the data contained in the PCB. Chapter 2 of the *MicroPower/Pascal Run-Time Services Manual* shows the layout of the PCB. After processing the exception, the handler must return the PCB to one of the kernel-managed state queues by using the RELEASE_EXCEPTION procedure.

Syntax

```
CONNECT_EXCEPTION (  DESC := queue-sem-descriptor
                     ⟦ GROUP := process-group ⟧
                     EXC_TYPE := exception-type
                     ⟦ STATUS := status-record ⟧ )
```

queue-sem-descriptor

The identifier of a variable of predefined type QUEUE_SEMAPHORE_DESC that contains the exception queue semaphore's structure identifier. The variable must have been previously initialized by a CREATE_QUEUE_SEMAPHORE or an INIT_STRUCTURE_DESC request.

process-group

A constant or the identifier of a variable of predefined type EXC_GROUP that contains a number between 0 and 255. That number identifies the group of processes for which the handler will service exceptions. The value 0 indicates that the handler will service exceptions for all process groups. The default value is 1.

exception-type

A constant or the identifier of a variable of predefined type EXC_SET that indicates the type(s) of exceptions to be received by this process (see Table 17–2).

status-record

The identifier of a variable of predefined record type EXC_STATUS that may receive an exception type and code. If you specify this parameter, the exception status, either success or error, that results from issuing the request is reported in this variable. Otherwise, an error causes the corresponding exception to be reported. The format of the exception record is described in Section 11.1.2.

Example

```
%INCLUDE 'EXC.PAS'

VAR
  MMU_queue : QUEUE_SEMAPHORE_DESC;

[PRIORITY(10), STACK_SIZE(100)] PROCESS MMU_handler;
BEGIN

(* Set up this process as the exception handler for memory faults. *)

  CONNECT_EXCEPTION
    (DESC := MMU_queue,
     GROUP := 3,
     EXC_TYPE := [MEMORY_FAULT]);

END; (* Process MMU_handler *)
```

Semantics

The CONNECT_EXCEPTION procedure makes, in the kernel's exception-dispatching table, an entry that describes a queue semaphore (identified by the sem-descriptor parameter) for a given combination of exception type and process group. Control then returns to the caller.

When an exception of the specified type for the specified group occurs, the kernel signals the handler's exception queue semaphore, puts the process causing the exception in the exception-wait state, and places its PCB on the handler's exception queue.

The group number permits several exception handlers for the same exception to coexist on the same system, each implementing a management strategy suited to a given class of processes. The group number 0 is a wildcard value specifying that the exception handler will be activated for exceptions that occur in all process groups.

If no handler exists for the group of process causing the exception, the kernel dismisses the exception and passes control to the exception procedure of the faulting process. If the faulting process has no exception procedure, the kernel aborts that process by forcing execution of its termination procedure. If no termination procedure was declared for that process, the OTS aborts the process.

The handler receives the exception by issuing a WAIT_EXCEPTION request on its exception queue semaphore. The handler subsequently processes the exception and releases the PCB with the RELEASE_EXCEPTION procedure. This procedure allows three courses of action: to abort the process, to pass the exception to the process's existing exception handler, or to return the process to the ready state, that is, to dismiss the exception.

This request is implemented through the CCND$ kernel primitive.

Error Returns

See Section 11.2 for general information about error returns. The following exception code may be returned:

ES$NMK (type: RESOURCE)—Resource not available; either the kernel's free-memory pool was exhausted (a table entry could not be allocated for the connection) or an entry already exists for the specified type/group combination

The request may return the following errors, though not as a result of standard Pascal programming practice:

ES$IAD	(type: SYSTEM_SERVICE)—Invalid address; pointer to structure is odd or not in user address space
ES$IPM	(type: SYSTEM_SERVICE)—Illegal parameter; either no bits were set in the mask word or more than one bit was set

Implementation Notes

You can use several calls to this procedure to connect one type of exception from several process groups to the same exception queue, if one exception-management strategy is applicable to several groups for an exception type. Alternatively, you can use a single call to this procedure to connect several different types of exceptions from one process group to the same exception queue.

Caution

If a handler itself causes an exception of a type that it handles, the handler process will lock up in a "fatal embrace" with itself and the process whose exception it was handling. (The handler will be blocked indefinitely on its own exception queue.) Exception handlers in general should not cause exceptions in a debugged application.

17.4 DISCONNECT_EXCEPTION

MACRO equivalent: CCND$

The DISCONNECT_EXCEPTION procedure disconnects a process from being the exception handler for an exception type and process group. DISCONNECT reverses the operation performed by the CONNECT_EXCEPTION procedure.

Syntax

```
DISCONNECT_EXCEPTION (  [[ GROUP := process-group ]]
                        EXC_TYPE := exception-type
                        [[ STATUS := status-record ]] )
```

process-group

A constant or the identifier of a variable of predefined type EXC_GROUP that contains a number between 0 and 255. That number identifies the group of processes for which this handler services exceptions. The default value is 1.

exception-type

A constant or the identifier of a variable of predefined type EXC_SET that indicates the type(s) of exceptions being disconnected (see Table 17–2).

status-record

The identifier of a variable of predefined record type EXC_STATUS that may receive an exception type and code. If you specify this parameter, the exception status, either success or error, that results from issuing the request is reported in this variable. Otherwise, an error causes the corresponding exception to be reported. The format of the exception record is described in Section 11.1.2.

Example

```
%INCLUDE 'EXC.PAS'

[PRIORITY(10), STACK_SIZE(100)] PROCESS MMU_handler;
BEGIN

  (* Remove the exception process for memory faults. *)
  DISCONNECT_EXCEPTION
    (GROUP := 3,
     EXC_TYPE := [MEMORY_FAULT]);

END; (* Process MMU_handler *)
```

Semantics

The DISCONNECT_EXCEPTION procedure removes from the kernel's exception-dispatching table the entry that describes a queue semaphore for a given combination of exception type and process group. Thereafter, when an exception of the specified type is caused by a process of the specified group, the kernel either passes the exception to the exception procedure of the faulting process, if one exists, or aborts that process by forcing execution of its termination procedure. See Chapter 6 of the *MicroPower/Pascal Run-Time Services Manual* for more information.

This request is implemented through the CCND$ kernel primitive.

Error Returns

See Section 11.2 for general information about error returns. The following exception code may be returned:

ES$NMK (type: RESOURCE)—Resource not available; either the kernel's free-memory pool was exhausted (a table entry could not be allocated for the connection) or an entry already exists for the specified type/group combination

The request may return the following errors, though not as a result of standard Pascal programming practice:

ES$IAD (type: SYSTEM_SERVICE)—Invalid address; pointer to buffer or structure is odd or not in user address space

ES$IPM (type: SYSTEM_SERVICE)—Illegal parameter; either no bits were set in the mask word or more than one bit was set

Implementation Notes

You may use a single call to this procedure to disconnect each type of exception managed by the same process or to disconnect multiple types from the same exception queue.

17.5 ESTABLISH

MACRO equivalent: SERA$

The ESTABLISH procedure declares a procedure to be the exception handler for a specified type of exception that may occur within its parent process. ESTABLISH allows a process to perform exception handling for a specified exception type.

Syntax

```
ESTABLISH ( EXC_PROCEDURE := procedure-identifier
            EXC_TYPE := exception-type )
```

procedure-identifier

The identifier of the procedure that is to process the exception. The procedure must be declared as described in Section 17.3.

exception-type

A constant or the identifier of a variable of predefined type EXC_SET that indicates the type(s) of exceptions to be processed by this procedure (see Table 17–2).

Restriction

An exception-handling procedure that is nested within another subprogram (another procedure, function, or process) should refer only to its own local variables or to statically allocated variables. ("Statically allocated" means allocated in memory rather than on the stack.) References to intermediate variables (nonlocal variables that are not statically allocated) cause unpredictable results. The compiler does not detect a failure to comply with this restriction.

Example

```
%INCLUDE 'EXC.PAS'

PROCEDURE Soft_io_and_Resource_errs
          (TYP : EXC_SET; COD : EXC_CODES;
           EXC_INFO_SIZE : UNSIGNED;
           VAR INFO : [READONLY] UNIVERSAL);
BEGIN
   .
   .
   .
END; (* Procedure Soft_io_and_Resource_errs *)

[INITIALIZE] PROCEDURE Init;
BEGIN

  (* Set up an exception procedure for soft i/o and resource exceptions. *)
  ESTABLISH
    (EXC_PROCEDURE := Soft_io_and_Resource_errs,
     EXC_TYPE := [SOFT_IO, RESOURCE]);

END; (* Procedure Init *)
```

Semantics

The ESTABLISH procedure connects a specified exception-handling procedure to the kernel's exception dispatcher. When an exception of the specified type occurs in this process, the kernel dismisses the exception and invokes the procedure if:

1. No exception-handling process was declared (CONNECT_EXCEPTION).
2. An exception-handling process used the RELEASE_EXCEPTION procedure's PASS option.

The exception-handling procedure may perform all steps necessary to manage the exception.

This request is implemented through the SERA$ kernel primitive.

Error Returns

See Section 11.2 for general information about error returns. The following exception codes may be returned:

ES$NMP	(type: RESOURCE)—Insufficient space for Pascal structure; not enough heap space for table entry

The request may return the following errors, though not as a result of standard Pascal programming practice:

ES$IAD	(type: SYSTEM_SERVICE)—Invalid address; pointer to structure is odd or not in user address space
ES$IPR	(type: SYSTEM_SERVICE)—Illegal primitive; request issued from ISR level

Implementation Notes

1. You may use a single call to this procedure to establish an exception procedure as the handler for each type of exception that it is to manage.
2. A subsequent call to this procedure with the same exception type parameter and a new procedure name parameter will establish the new procedure as the handler for that exception type. (An implicit REVERT is thus performed for the procedure that previously handled this exception type.)
3. If an exception-handling or terminate procedure is implemented to handle exceptions resulting from Pascal I/O statements (for example, illegal integer values), the exception-handling or terminate procedure should not perform I/O. The indivisible nature of a Pascal I/O statement is inconsistent with the asynchronous characteristic of an exception handler or a terminate procedure.

17.6 RELEASE_EXCEPTION

MACRO equivalent: DEXC$

The RELEASE_EXCEPTION procedure returns a process in the exception-wait state to the kernel for further disposition. The process's state is changed to the appropriate ready state.

This procedure allows an exception-handler process to dispose of an exception that it has received (after processing the exception and determining a course of action). An action option specified in the call directs the kernel to dismiss the exception, abort the process, or pass the exception to the process's exception-handling procedure, if any.

The ESTABLISH and REVERT requests allow a process to declare its own internal exception-handling procedures.

Syntax

$$\text{RELEASE\_EXCEPTION (}\quad \text{ACTION} := \left\{\begin{array}{l}\text{DISMISS}\\ \text{ABORT}\\ \text{PASS}\end{array}\right\}$$
$$\text{PCB\_PTR} := \text{pcb-pointer)}$$

ACTION

Indicates, by one of the following options, the action to take in disposing of the exception.

Option	Description
DISMISS	Dismiss the exception and place the process in the ready state.
ABORT	Abort the process by forcing execution of its termination procedure.
PASS	Pass the exception to the process, if it has an exception-handling procedure defined within it; otherwise, abort the process.

pcb-pointer

The identifier of a variable of predefined type PCB_POINTER that contains the pointer to the PCB of the process that caused the exception (ordinarily obtained through the WAIT_EXCEPTION request).

Example

```
%INCLUDE 'EXC.PAS'
%INCLUDE 'PCBM.PAS'

VAR
  MMU_queue : QUEUE_SEMAPHORE_DESC;
  Pcbptr : PCB_POINTER;

[PRIORITY(10), STACK_SIZE(100)] PROCESS MMU_handler;
BEGIN

  (* Wait for an exception. *)
  WAIT_EXCEPTION
    (PCB_PTR := Pcbptr,
     DESC := MMU_queue);
  (* Pass the exception. *)
  RELEASE_EXCEPTION
    (ACTION := PASS,
     PCB_PTR := Pcbptr);
END; (* Process MMU_handler *)
```

Semantics

The RELEASE_EXCEPTION procedure places the PCB of the process, which was received in the exception-wait-active state by means of the caller's exception queue, on the appropriate ready-state queue for disposition as requested in the call. (The PCB is placed on the ready-active queue unless it was suspended while in the exception-wait state.) The kernel will do the following:

1. Cancel the exception, allowing the process to be reentered normally when it is rescheduled (ACTION := DISMISS).
2. Abort the process, causing its termination procedure to be executed when the process is rescheduled (ACTION := ABORT).
3. Cancel the exception and pass control to the process's own exception procedure. If no such procedure exists or if the process has not requested handling of the type of exception, its termination procedure will be executed instead (ACTION := PASS).

This request is implemented through the DEXC$ kernel primitive.

Error Returns

See Section 11.2 for general information about error returns. The request may return the following error, though not as a result of standard Pascal programming practice:

ES$IPM	(type: SYSTEM_SERVICE)—Illegal parameter; invalid PCB address or an illegal action code

Implementation Notes

To use this procedure, you must select the appropriate PCB declaration file as described in Appendix I.

17.7 REPORT

MACRO equivalent: REXC$

The REPORT procedure declares an exception to the kernel to allow a process to announce a self-detected software exception or to simulate a hardware exception. See Chapter 6 of the *MicroPower/Pascal Run-Time Services Manual* for more information.

Syntax

```
REPORT ( [[ EXC_INFO := info-data ]]
         [[ EXC_INFO_SIZE : info-data-length ]]
         EXC_CODE := exception-code
         EXC_TYPE := exception-type )
```

info-data

The identifier of a variable of any user-specified data type. This parameter passes exception-specific data in the optional argument list portion of the reporting process's exception stack frame to an exception-handling procedure. If you specify 0 or omit this parameter, you must omit the info-data-length parameter also. The default value is 0. Chapter 6 of the *MicroPower/Pascal Run-Time Services Manual* provides the general format of the exception stack frame and the exception-specific argument lists supplied by various hardware exceptions.

info-data-length

A constant or the identifier of a variable of type UNSIGNED that specifies the number of bytes of information provided in the variable specified by the info-data parameter. The value specified must be an even number. If you specify 0 or omit this parameter, you must omit the info-data parameter also. The default value is 0.

Caution

When calculating the byte-count value for the info-data-length parameter, use SIZE function to minimize the possibility of error. An incorrect byte-count value will produce unpredictable and catastrophic effects on application operation.

exception-code

A constant or the identifier of a variable of predefined type EXC_CODES that contains the exception code for the exception being reported (see Table 17–2).

exception-type

A constant or the identifier of a variable of predefined type EXC_SET that indicates the type(s) of exceptions being reported (see Table 17–2).

Example

```
%INCLUDE 'EXC.PAS'

PROCEDURE P1;
  VAR
    I : INTEGER;
  BEGIN

  (* Report an illegal integer exception. *)
  REPORT
    (EXC_TYPE := [SOFT_IO],
     EXC_CODE := ES$IIV,
     EXC_INFO_SIZE := SIZE(I),
     EXC_INFO := I);

END; (* Procedure P1 *)
```

Semantics

The REPORT procedure announces an exception to the kernel, which performs one of the following actions:

1. If an exception-handling process exists for this exception, the kernel dispatches the exception to the handling process and places the reporting process in the exception-wait state.
2. If no exception-handling process exists for this exception, the exception is dispatched to the exception-handling procedure, if one exists, of the reporting process.
3. If neither an exception-handling process nor an exception-handling procedure exists, the kernel forces execution of the termination procedure, if one exists, of the reporting process.
4. If none of these conditions is met, the kernel aborts the reporting process.

This request is implemented through the REXC$ kernel primitive.

Error Returns

See Section 11.2 for general information about error returns. The request may return the following errors, though not as a result of standard Pascal programming practice:

ES$IAD	(type: SYSTEM_SERVICE)—Invalid address; pointer to argument buffer is odd or not in user address space
ES$IPM	(type: SYSTEM_SERVICE)—Illegal parameter; either no bits were set in the mask word, more than one bit was set, or the argument buffer length value was odd

17.8 REVERT

MACRO equivalent: SERA$

The REVERT procedure releases the specified exception-handling procedure from responding to occurrences of the specified exception within the parent process. This procedure reverses the operation performed by the ESTABLISH procedure.

Syntax

REVERT (EXC_TYPE := exception-type)

exception-type

A constant or the identifier of a variable of predefined type EXC_SET that contains the type(s) of exceptions being released (see Table 17–2).

Example

```
%INCLUDE 'EXC.PAS'

[TERMINATE] PROCEDURE Term;
BEGIN

  (* Remove the exception procedure for soft i/o exceptions. *)
  REVERT
    (EXC_TYPE := [SOFT_IO]);

END; (* Procedure Term *)
```

Semantics

The REVERT procedure breaks the connection between the specified exception-handling procedure and the kernel's exception dispatcher for exceptions of the specified type.

After REVERT is invoked, occurrence of an exception of the specified type within the parent process will cause the exception dispatcher to do one of the following:

- Dispatch the exception to an exception-handling process, if one exists for this exception type
- Force execution of the parent process's termination procedure, if there is no exception-handling process for this exception type
- Abort the parent process if no termination procedure exists

This request is implemented through the SERA$ kernel primitive.

Error Returns

See Section 11.2 for general information about error returns. The following exception code may be returned:

ES$EPN (type: SYSTEM_SERVICE)—Exception procedure not defined

The request may return the following error, though not as a result of standard Pascal programming practice:

ES$IPR (type: SYSTEM_SERVICE)—Illegal primitive; request issued from ISR level

Implementation Notes

You may use a single call to this procedure for all exception types being released.

17.9 WAIT_EXCEPTION

MACRO equivalent: WAIQ$

The WAIT_EXCEPTION procedure permits an exception-handling process to be notified that an exception occurred. The procedure waits on a specified exception queue semaphore for a process control block (PCB). When it becomes available, WAIT_EXCEPTION removes the PCB from the semaphore's queue and returns the PCB's pointer to the caller. If no PCB is available, the calling process is blocked on the semaphore, until an exception occurs.

This procedure is for use by processes with the PRIVILEGED or DRIVER attributes or by processes that reside in an unmapped-memory environment (see Section 10.1.2).

Note

> If a process with general mapping attempts to access the contents of a PCB, the results may be unpredictable. The kernel will generate a memory-management exception (MS$MMU) so long as the PCB's address is not also a valid address in the process's address space. No exception will occur if the PCB's address is also a valid address in the process's space, and the process will then obtain invalid data.

The RELEASE_EXCEPTION request is the complement of the WAIT_EXCEPTION request and places the PCB of the process on the appropriate ready-state queue for disposition as requested in the call.

Syntax

```
WAIT_EXCEPTION ( PCB_PTR := pcb-pointer
                 { DESC := queue-sem-descriptor }
                 { NAME := queue-sem-name       } )
```

pcb-pointer

The identifier of a variable of predefined type PCB_POINTER that will receive the pointer to the PCB of the process that created the exception.

queue-sem-descriptor

The identifier of a variable of predefined type QUEUE_SEMAPHORE_DESC that contains the exception queue semaphore's structure identifier. The variable must have been previously initialized by a CREATE_QUEUE_SEMAPHORE or an INIT_STRUCTURE_DESC request.

queue-sem-name

A character-string constant or the identifier of a variable of predefined type NAME_STR that specifies the 6-character name of an existing exception queue semaphore (see Section 11.1.1.1).

Example

```
%INCLUDE 'EXC.PAS'
%INCLUDE 'PCBM.PAS'

VAR
  MMU_queue : QUEUE_SEMAPHORE_DESC;
  Pcbptr : PCB_POINTER;

[PRIORITY(10), STACK_SIZE(100)] PROCESS MMU_handler;
BEGIN

  (* Wait for an exception. *)
  WAIT_EXCEPTION
    (PCB_PTR := Pcbptr,
     DESC := MMU_queue);

END; (* Process MMU_handler *)
```

Semantics

The WAIT_EXCEPTION procedure tests the specified exception queue semaphore for an available PCB. If at least one PCB is on the semaphore's queue, the procedure does the following:

1. Decrements the specified queue semaphore
2. Removes the first available PCB from the queue
3. Returns the pointer to that PCB in the variable specified by the pcb-pointer parameter

If there is no PCB available on the queue, the request blocks the calling process and calls the scheduler. The calling process remains blocked until it can be reactivated by a subsequent signal of the semaphore, which places a PCB on the queue.

This request is implemented through the WAIQ$ kernel primitive.

Error Returns

See Section 11.2 for general information about error returns. The following exception codes may be returned:

ES$CDN	(type: SYSTEM_SERVICE)—Cannot specify both descriptor and name
ES$IST	(type: SYSTEM_SERVICE)—Invalid structure descriptor; exception queue semaphore does not exist
ES$MDN	(type: SYSTEM_SERVICE)—Must specify descriptor or name

Chapter 18
Dynamic Memory-Allocation and Region-Sharing Requests

This chapter describes the requests that let a process control dynamically the allocation and mapping of target memory. These requests, implemented through the predeclared procedures and functions listed in Table 18–1, are the Pascal language interface to the services provided by the kernel's memory-allocation and region-sharing primitives. The requests collectively let your processes:

1. Obtain an area of unused memory and, optionally, release it after temporary use (dynamic memory allocation/deallocation)
2. Share an area of "static" or "dynamic" memory between static process families (region sharing)
3. In a mapped system, obtain a virtual-address window into either a dynamic or shared memory area, in support of capabilities 1 and 2 (dynamic mapping)

These related capabilities are intended principally to support large memory configurations and shared-common memory in mapped target systems and are described here in that context unless otherwise indicated. Dynamic RAM allocation may be useful in some unmapped applications, but memory sharing by using kernel primitives has a very limited utility, since more efficient design alternatives exist in the unmapped environment.

Refer to Chapter 5 of the *MicroPower/Pascal Run-Time Services Manual* for a discussion of memory-allocation and region-sharing concepts. The chapter also shows the relationship between the Pascal requests and the kernel's (MACRO–11) primitive service requests and includes pertinent coding examples.

You must include in your program or module the definitions of the routines that implement the requests described in this chapter before using them. See Appendix I for more information.

Table 18–1 summarizes the Pascal dynamic memory-allocation, region-sharing, and dynamic mapping requests.

Table 18-1: Memory-Allocation and Region-Sharing Requests

Request	Operation
Dynamic Memory-Allocation	
ALLOCATE_REGION	Allocates an area of unused physical memory to the calling process
DEALLOCATE_REGION	Allows the calling process to return a physical memory region, previously acquired by an ALLOCATE_REGION request
Region-Sharing	
CREATE_SHARED_REGION	Allows the calling process to declare a region of memory to be shareable by other programs and to assign a systemwide runtime name to the region
ACCESS_SHARED_REGION	Allows the calling process to gain access, through the runtime name assigned to the shared region, to a region of memory that was previously made shareable by another process
DELETE_SHARED_REGION	Allows the calling process to delete the shared region descriptor (SRD) identified in the call to preclude any subsequent access to the region by using the ACCESS_SHARED_REGION request
Dynamic Mapping	
MAP_WINDOW	Permits a process to associate a window of virtual addresses with a specified region of physical memory
UNMAP_WINDOW	Permits a process to reverse the effect of a prior MAP_WINDOW operation, dissociating a sequence of virtual addresses—the virtual window—from the physical memory to which it was mapped
SAVE_CONTEXT	Permits a process to save a copy of its current memory mapping for subsequent restoration by the RESTORE_CONTEXT request
RESTORE_CONTEXT	Permits a process to reset itself to an earlier state of virtual-to-physical mapping previously saved by the SAVE_CONTEXT request
GET_MAPPING	Allows the calling process to obtain a copy of its own current mapping or that of any other specified process

18.1 ACCESS_SHARED_REGION

MACRO equivalent: ACSR$

The ACCESS_SHARED_REGION procedure uses the run-time name assigned to the shared region to allow the calling process to gain access to a region of memory that was previously made shareable by another process. (It is also possible to access a shared region that was defined at build time by a MEMORY configuration macro.) More precisely, the ACCESS_SHARED_REGION procedure returns a physical description of the named shared region to a region ID block (RIB) that is specified in the call. The RIB information is normally used in a subsequent window-mapping operation, performed for general-mapped processes by the MAP_WINDOW request.

The accessed region can be either a common or a physical shared region (see the CREATE_SHARED_REGION request). The information returned in the RIB describes the region's location, size, and mode attribute.

Although region sharing by using kernel services is applicable primarily to a mapped target environment, the CREATE_SHARED_REGION and the ACCESS_SHARED_REGION requests can be useful in an unmapped application containing more than one user program. However, because of the single address space in an unmapped system, no advantage is generally gained from having multiple user programs. Coding details differ for unmapped usage, since no distinction is made between virtual and physical addresses. In the unmapped case, the RIB always specifies the base of the region directly as a physical address, and the region size is represented in bytes. The base and size information supplied in the RIB is used directly. Also, common and physical regions are effectively equivalent; the region offset is always 0.

A semaphore is usually required to protect against concurrent references to a region shared by several processes. Also, the kernel structure, SRD, that represents a shared region can be deleted by using the DELETE_SHARED_REGION request, though typically that is done only if the creating program terminates. The kernel does not provide any automatic safeguard against inadvertent reference to a deleted (and possibly deallocated) shared region, since any process that accessed the region while shareable retains a description of it.

Chapter 5 of the *MicroPower/Pascal Run-Time Services Manual* discusses region sharing, including the use of ACCESS_SHARED_REGION in the context of other related requests. The CREATE_SHARED_REGION request provides the complementary create operation, which declares a region as being shareable and assigns its run-time name.

Syntax

```
ACCESS_SHARED_REGION (  RIB := region-id-block
                        { DESC := structure-desc }
                        { NAME := region-name    }
                        [[ STATUS := status-record ]] )
```

region-id-block

The identifier of a variable of predefined type REGION_ID_BLOCK that contains a RIB in which the location, size, and mode attribute of the region to be made shareable are returned by the request, as described under Semantics.

structure-descriptor

The identifier of a variable of predefined type STRUCTURE_DESC that contains the shared region's structure identifier. The variable must have been previously initialized by a CREATE_SHARED_REGION request.

region-name

A character-string constant or the identifier of a variable of predefined type NAME_STR that contains the 6-character name of an existing shared region (see Section 11.1.1.1).

status-record

The identifier of a variable of predefined record type EXC_STATUS that may receive an exception type and code. If you specify this parameter, the exception status, either success or error, that results from issuing the request is reported in this variable. Otherwise, an error causes the corresponding exception to be reported. The format of the exception record is described in Section 11.1.2.

Example

```
%INCLUDE 'DRAM.PAS'

VAR
  Rib_1 : REGION_ID_BLOCK;

[PRIORITY(10), STACK_SIZE(100)] PROCESS A;
BEGIN

  (* Access a shared region. *)
  ACCESS_SHARED_REGION
    (RIB := Rib_1,
     NAME := 'REG1  ');

END; (* Process A *)
```

Semantics

ACCESS_SHARED_REGION looks for the kernel data structure SRD having the name specified in the region-name parameter or having the structure identifier specified in the structure-descriptor parameter. If that SRD exists, the request copies information contained in the SRD to the RIB specified by the region-id-block parameter and returns to the caller. If no such SRD exists, the request returns to the caller, with an error indication.

Information describing the accessed region is returned to the user in the variable specified by the region-id-block parameter. The variable is of the predefined type as defined in the DRAM.PAS %INCLUDE file.

```
                                            {Mode}
                               {Physical    Common      Unmapped}
REGION_ID_BLOCK = RECORD
  REGION_ADDRESS : UNIVERSAL;  {PAR value / PAR value / phys. address}
  REGION_SIZE : UNSIGNED;      {PAR ticks / PAR ticks / bytes        }
  REGION_MODE : ADDRESS_TYPE;  {PHYSICAL  / COMMON    / -------      }
  REGION_OFFSET : UNSIGNED;    {0         / bytes     / 0            }
  END;
```

REGION_ADDRESS

In a mapped environment, the region's base address, always on a 32-word physical boundary, returned as a physical PAR value (unsigned integer). (That value is not directly usable as an address, of course, but can be used in a physical-to-virtual mapping operation as provided by the MAP_WINDOW request.)

In an unmapped environment, the region base is a physical address that can be used directly.

REGION_SIZE

In a mapped environment, the number of PAR ticks (units of 32 words) in the region. In the case of a common region, the described size represents the actual size of the region—specified to CREATE_SHARED_REGION in bytes—rounded up to the next multiple of 32 words.

In an unmapped environment, the region size is the number of bytes specified in the CREATE_SHARED_REGION request.

REGION_MODE

The enumerated type values COMMON and PHYSICAL, denoting a common or physical region, respectively.

REGION_OFFSET

Relevant only for a shared common region, an increment, in bytes, from the PAR value to the beginning of the region. (The region-offset field is significant for the MAP_WINDOW operation.)

In an unmapped environment, the region offset is always 0, regardless of the region mode.

This request is implemented through the ACSR$ kernel primitive.

Error Returns

See Section 11.2 for general information about error returns. The following exception codes may be returned:

ES$CDN	(type: SYSTEM_SERVICE)—Cannot specify both descriptor and name
ES$IST	(type: SYSTEM_SERVICE)—Invalid structure descriptor; shared region does not exist
ES$MDN	(type: SYSTEM_SERVICE)—Must specify descriptor or name

The request may return the following error, though not as a result of standard Pascal programming practice:

ES$IAD	(type: SYSTEM_SERVICE)—Invalid address; odd or not in user space

18.2 ALLOCATE_REGION

MACRO equivalent: ALRG$

The ALLOCATE_REGION function allocates an area of unused physical memory, if available, to the calling process. The memory area, called a region, is of user-specified size and is allocated dynamically from a list of free RAM segments maintained by the kernel. (See Chapter 5 of the *MicroPower/Pascal Run-Time Services Manual*.) If a region is successfully allocated, the function returns control to the calling process with a Boolean TRUE value and other information. If a region of the required size cannot be allocated, the function returns control to the caller with a Boolean FALSE value.

Allocation is achieved through a user-supplied RIB, in which the function returns information about the location and size of the allocated region. The process that owns the RIB is responsible for the region and can use it for any purpose; the kernel does not keep track of the allocated space. When the space is no longer needed, you can deallocate a physical region by using the DEALLOCATE_REGION request.

Although dynamic RAM allocation is designed primarily for a mapped target environment, the ALLOCATE_REGION request can be used in an unmapped application as well. Coding details differ between mapped and unmapped usage. In the mapped case, the caller specifies the required region size in terms of PAR ticks, that is, in units of 32-word blocks (100 octal bytes). The function returns the physical base address of the region as a PAR value and returns the region size in PAR ticks. This PAR information, returned in the RIB, can be used in subsequent window-mapping operations by using the MAP_WINDOW request.

In the unmapped case, the caller specifies the required region size directly in bytes. The function returns the base address of the region directly, of course, and returns the region size, in bytes, rounded up to the next multiple of 4, if necessary.

Chapter 5 of the *MicroPower/Pascal Run-Time Services Manual* discusses dynamic RAM allocation, including the use of ALLOCATE_REGION in the context of other related requests.

Syntax

```
ALLOCATE_REGION ( RIB := region-id-block
                  REG_SIZE := region-size )
```

region-id-block

The identifier of a variable of predefined type REGION_ID_BLOCK in which the location, size, and mode attribute of the region are returned by the request, as described under Semantics. (The mode of a dynamically allocated region is always PHYSICAL.)

region-size

A constant or the identifier of a variable of predefined type UNSIGNED that specifies the size of the region to be allocated. For a mapped application, the size value specifies the number of 32-word (100(octal)-byte) blocks required. For an unmapped application, the size value specifies the number of bytes required.

Example

```
%INCLUDE 'DRAM.PAS'

VAR
  Rib_1 : REGION_ID_BLOCK;
  Allocated : BOOLEAN;

[PRIORITY(10), STACK_SIZE(100)] PROCESS A;
BEGIN

  (* Allocate a region. *)
  Allocated := ALLOCATE_REGION
                 (RIB := Rib_1,
                  REG_SIZE := %O'200');

END; (* Process A *)
```

Semantics

The ALLOCATE_REGION function checks the kernel's free RAM list for a memory segment that equals or exceeds the size of the requested region. If such a segment exists, the function removes the required amount of memory from the free RAM list, modifies the caller's RIB area and returns control to the caller with a Boolean TRUE value. (The ALLOCATE_REGION allocates from the free RAM list on a first-fit basis.) If no sufficiently large free RAM segment exists, the ALLOCATE_REGION returns control to the calling process, with a Boolean FALSE value.

The information describing the accessed region is returned to the user in the variable specified by the region-id-block parameter. The variable is of the predefined type as defined in the DRAM.PAS %INCLUDE file.

```
                                            {Mode}
                                  {Physical    Unmapped}
REGION_ID_BLOCK = RECORD
  REGION_ADDRESS : UNIVERSAL;     {PAR value / phys. address}
  REGION_SIZE : UNSIGNED;         {PAR ticks / bytes        }
  REGION_MODE : ADDRESS_TYPE;     {PHYSICAL  / -------      }
  REGION_OFFSET : UNSIGNED;       {0         / 0            }
  END;
```

REGION_ADDRESS

In a mapped environment, the region's base address, always on a 32-word physical boundary, returned as a physical PAR value (unsigned integer). (That value is not directly usable as an address, of course, but can be used in a physical-to-virtual mapping operation as provided by the MAP_WINDOW request.)

In an unmapped environment, the region address is a physical address that can be used directly.

REGION_SIZE

In a mapped environment, an integer representing the number of PAR ticks (32-word blocks) allocated, as represented in the allocation request.

In an unmapped environment, the region size is an integer representing the number of bytes allocated. If the requested number of bytes was not a multiple of 4, the next higher multiple of four bytes is allocated.

REGION_MODE

In both mapped and unmapped environments, a value returned as PHYSICAL. The mode of a region, PHYSICAL or COMMON, is significant to the CREATE_SHARED_REGION request and, indirectly, to the MAP_WINDOW request.

REGION_OFFSET

A field significant only in operations on shared common regions. A value of 0 is always returned, as appropriate for a physical region.

This request is implemented through the ALRG$ kernel primitive.

Error Returns

See Section 11.2 for general information about error returns. The request may return the following error, though not as a result of standard Pascal programming practice:

ES$IAD	(type: SYSTEM_SERVICE)—Invalid address; the RIB address is not on a word boundary

18.3 CREATE_SHARED_REGION

MACRO equivalent: CRSR$

The CREATE_SHARED_REGION procedure allows the calling process to declare a region of memory to be shareable by other static processes and to assign a system-wide run-time name to the region. More precisely, CREATE_SHARED_REGION creates a named kernel data structure called a shared region descriptor (SRD) that describes the memory region specified by the caller. Subsequently, other processes can gain access to the shared region by using the ACCESS_SHARED_REGION request, by means of the run-time name associated with the SRD.

Note

CREATE_SHARED_REGION is relevant primarily to a mapped memory environment and is described in terms of a mapped application except where indicated otherwise.

A shared region can be either a shared common region or a shared physical region. A common region exists within the caller's statically allocated address space; the location of a shared common region is therefore determined by the process declaring it as shared. A physical region was dynamically allocated from unused physical memory by an allocate-region operation. Thus, the location of a shared physical region is initially determined by the ALLOCATE_REGION request. Chapter 5 of the *MicroPower/Pascal Run-Time Services Manual* discusses common versus physical regions.

Whether common or physical, the region to be made shareable is identified by a RIB in user space that is specified in the call. The RIB specifies the region's location, size, and mode attribute. The location, or base, of a common region is specified as a virtual address, the size is specified in bytes, and the mode attribute is COMMON. The information describing a common region is placed in the RIB by the user process. The procedure modifies the information supplied in the caller's RIB for a common region, replacing the virtual description with a physical description, as described under Semantics. Normally, the information in the RIB for a physical region is precisely that returned by the prior ALLOCATE_REGION call that allocated the region.

Although region sharing by using kernel services is applicable primarily to a mapped target environment, CREATE_SHARED_REGION can be useful in an unmapped application containing more than one user static process. However, because of the single address space in an unmapped system, no advantage is generally gained from having multiple user static processes. Coding details differ for unmapped usage, since no distinction is made between virtual and physical addresses. The RIB always specifies the base of a region directly as a physical address, and the region size is represented in bytes. Therefore, the distinction between common and physical region is not significant for unmapped shared-region creation, although the COMMON and PHYSICAL mode attributes are recognized and applied to the SRD and should be used consistently.

Chapter 5 of the *MicroPower/Pascal Run-Time Services Manual* discusses region sharing, including the use of CREATE_SHARED_REGION in the context of other related requests.

The ACCESS_SHARED_REGION request provides the complementary access operation, which returns RIB information based on a specified shared region name.

Syntax

```
CREATE_SHARED_REGION (  RIB := region-id-block
                        [[ DESC := structure-desc ]]
                        NAME := region-name
                        [[ STATUS := status-record ]] )
```

region-id-block

The identifier of a variable of predefined type REGION_ID_BLOCK that contains the location, size, and mode attribute of the region to be made shareable, as described under Semantics. In a mapped environment, the region's address, size, and offset fields for a common region are modified by the request. That is, the RIB is both a source and a destination variable in the mapped common case.

structure-descriptor

The identifier of a variable of predefined type STRUCTURE_DESC that is to receive the shared region's structure identifier.

region-name

A character-string constant or the identifier of a variable of predefined type NAME_STR that contains the 6-character name of the region to be made shareable (see Section 11.1.1.1). The name must not be the name of an existing process or structure.

status-record

The identifier of a variable of predefined record type EXC_STATUS that may receive an exception type and code. If you specify this parameter, the exception status, either success or error, that results from issuing the request is reported in this variable. Otherwise, an error causes the corresponding exception to be reported. The format of the exception record is described in Section 11.1.2.

Example

```
%INCLUDE 'DRAM.PAS'

TYPE
  Common_1 = ARRAY [1..%O'10000'] OF INTEGER;

VAR
  Rib_1 : REGION_ID_BLOCK;
  Reg_1 : Common_1;

[PRIORITY(10), STACK_SIZE(100)] PROCESS A;
BEGIN

  (* Create a shared region. *)
  WITH Rib_1 DO
   BEGIN
     NEW (Reg_1);
     REGION_ADDRESS := Reg_1 :: UNSIGNED;
     REGION_OFFSET := 0;
     REGION_SIZE := %O'20000';
     REGION_MODE := COMMON;
     CREATE_SHARED_REGION
       (RIB := Rib_1,
        NAME := 'REG1  ');
   END;

END; (* Process A *)
```

Semantics

The CREATE_SHARED_REGION procedure creates a SRD structure in the kernel's system-common area, using the region's base address and size information contained in the RIB specified by the caller. The request associates the name specified by the region-name parameter with the SRD structure and returns the structure identifier of the SRD in the variable specified by the structure-desc parameter.

The information describing the shared region specified by the region-id-block parameter is of the predefined type as defined in the DRAM.PAS %INCLUDE file.

```
                                         {Mode}
                                {Physical    Common}
REGION_ID_BLOCK = RECORD
  REGION_ADDRESS : UNIVERSAL;   {PAR value / virtual address}
  REGION_SIZE : UNSIGNED;       {PAR ticks / bytes          }
  REGION_MODE : ADDRESS_TYPE;   {PHYSICAL  / COMMON         }
  REGION_OFFSET : UNSIGNED;     {          ignored          }
  END;
```

REGION_ADDRESS

In a mapped environment, the region's virtual address if the REGION_MODE is COMMON. On return to the caller, the address is converted to a nearest physical PAR value. If the REGION_MODE is PHYSICAL, no conversion is required, since the value provided by the prior ALLOCATE_REGION call is already in the appropriate form (PAR value).

In an unmapped environment, this field contains a physical address regardless of the region mode.

REGION_SIZE

In a mapped environment, the number of bytes in the region if the REGION_MODE is COMMON. On return to the caller, this value is converted to the number of PAR ticks (32-word units) in the region. If the byte value you supply is not a multiple of 32, the result is rounded up to the next multiple of 32. If the REGION_MODE is PHYSICAL, no conversion is required, since the value is already in the appropriate form (PAR ticks).

In an unmapped environment, this field contains the number of bytes in the region; no conversion is necessary.

REGION_MODE

The enumerated type values COMMON and PHYSICAL, denoting a common or physical region respectively, although no effective distinction is made between the two in the unmapped case.

REGION_OFFSET

A value indicating the positive displacement in bytes, if any, of the common region base address from the calculated PAR value. The field is significant only in shared common-region mapping operations. Effectively, the offset field value is assumed to be 0 for all operations on mapped physical regions as well as for all unmapped operations.

This request is implemented through the CRSR$ kernel primitive.

Error Returns

See Section 11.2 for general information about error returns. The following exception codes may be returned:

ES$MSS	(type: SYSTEM_SERVICE)—Must specify structure descriptor; the structure-desc parameter is not optional
ES$NMK	(type: RESOURCE)—Insufficient space for kernel structure; could not create shared region
ES$SNI	(type: SYSTEM_SERVICE)—Structure name in use; a kernel structure already exists with the name you specified for the region

The request may return the following error, though not as a result of standard Pascal programming practice:

ES$IAD	(type: SYSTEM_SERVICE)—Invalid address; the RIB address is not on a word boundary

18.4 DEALLOCATE_REGION

MACRO equivalent: DLRG$

The DEALLOCATE_REGION procedure allows the calling process to return a physical memory region, previously allocated by ALLOCATE_REGION, to the list of free RAM segments maintained by the kernel. (See Chapter 5 of the *MicroPower/Pascal Run-Time Services Manual.*) The base, size, and mode of the region to be deallocated are specified by a RIB in the caller's address space. The procedure zeroes the size field contained in the RIB on successful deallocation. The mode of the region must be PHYSICAL.

DEALLOCATE_REGION attempts to consolidate the free RAM list whenever possible by combining the newly deallocated space with any adjoining space already represented in the list. Such consolidation results in a new free segment that is larger than the region just deallocated.

Whether or not list consolidation takes place, any region deallocation may free up enough space to allow a previously unsuccessful allocation request issued by another process to be satisfied if the request were reissued. DEALLOCATE_REGION always returns control to the calling process.

Although dynamic RAM allocation is designed primarily for a mapped target environment, you can use the ALLOCATE_REGION and DEALLOCATE_REGION requests in an unmapped application as well. RIB content differs between mapped and unmapped usage, as described for ALLOCATE_REGION. (Presumably, the RIB supplied to DEALLOCATE_REGION contains the values that were returned by a call to ALLOCATE_REGION.)

Chapter 5 of the *MicroPower/Pascal Run-Time Services Manual* discusses dynamic RAM allocation and deallocation in the context of other related requests.

Syntax

DEALLOCATE_REGION (RIB := region-id-block)

region-id-block

The identifier of a variable of predefined type REGION_ID_BLOCK that contains the location, size, and mode attribute of the region to be deallocated, as described under Semantics.

Example

```
%INCLUDE 'DRAM.PAS'

VAR
  Rib_1 : REGION_ID_BLOCK;

[PRIORITY(10), STACK_SIZE(100)] PROCESS A;
BEGIN

  (* Deallocate a region. *)
  DEALLOCATE_REGION
    (RIB := Rib_1);

END; (* Process A *)
```

Semantics

The DEALLOCATE_REGION procedure adds the memory space described by the caller's RIB to the kernel's linked list of free RAM segments, either by inserting a new list element or by modifying an existing list element. (As a consequence, the information in the user's RIB is no longer valid.) If the region-size field is nonzero, DEALLOCATE_REGION zeroes the size field, deallocates the described region, and returns control to the calling process. Otherwise, the procedure returns to the caller, with an error indication.

The information describing the region to deallocate resides in the variable specified in the region-id-block parameter and must be of the same form as that returned by a corresponding region-allocation operation. The variable is of the type as defined in the DRAM.PAS %INCLUDE file.

```
                                         {Mode}
                                {Mapped       Unmapped}
REGION_ID_BLOCK = RECORD
  REGION_ADDRESS : UNIVERSAL;   {PAR value / phys. address}
  REGION_SIZE : UNSIGNED;       {PAR ticks / bytes        }
  REGION_MODE : ADDRESS_TYPE;   {PHYSICAL  / -------      }
  REGION_OFFSET : UNSIGNED;     {         ignored         }
  END;
```

REGION_ADDRESS

In a mapped memory environment, a value representing a 32-word physical boundary: the region's physical PAR value.

In an unmapped memory environment, this field contains the physical address of the region to be deallocated.

REGION_SIZE

In a mapped memory environment, the number of consecutive 32-word units (PAR ticks) to be deallocated starting at the region base address.

In an unmapped memory environment, the field contains the number of bytes to be deallocated starting at the region base. If the specified size is not a multiple of 4, the next higher multiple of four bytes is deallocated.

On return from the caller, the field contains zeros if the request was successful.

REGION_MODE

The enumerated type value PHYSICAL, denoting a physical region.

REGION_OFFSET

Significant only in operations on shared common regions.

This request is implemented through the DLRG$ kernel primitive

Error Returns

See Section 11.2 for general information about error returns. The following exception code may be returned:

ES$IPM (type: SYSTEM_SERVICE)—Illegal parameter; the region-size value in the RIB is 0. The region is already deallocated

The request may return the following error, though not as a result of standard Pascal programming practice:

ES$IAD (type: SYSTEM_SERVICE)—Invalid address; the RIB address is not on a word boundary

Implementation Notes

The DEALLOCATE_REGION procedure does not limit you to deallocating an entire region, as originally allocated, in a given operation. You can deallocate just a portion of a region or return a region piecemeal in successive operations. Partial deallocation, which might be useful in some applications, entails user modification of supplied RIB contents: the region base and size values supplied by ALLOCATE_REGION. To avoid obscure run-time problems, considerable care should be taken to ensure the correctness of any such modifications, since the request does minimal checking of RIB values. Any deallocation error introduced by user-modified values will corrupt the kernel's free RAM list with unpredictable consequences—typically a delayed system crash. The integrity of the free RAM list depends entirely on the validity of the space descriptions supplied in deallocation requests.

18.5 DELETE_SHARED_REGION

MACRO equivalent: DLST$

The DELETE_SHARED_REGION procedure lets the calling process delete the SRD identified in the call (that is, the kernel data structure that represents a region as shared). The effect of the operation is to preclude any subsequent access to the region by using the ACCESS_SHARED_REGION request. However, the operation does not disable any previously gained access to the region.

Typically, DELETE_SHARED_REGION would be used only in the termination routine of the process responsible for creating the SRD. (Processes commonly delete structures they have created if forced to terminate.) The kernel does not provide any automatic safeguard against inadvertent reference to a deleted (and possibly deallocated) shared region, since any process that previously accessed the region while it was shareable retains a description of it. The effective lifetime of a shared region could be coordinated among the processes having access to it through a special semaphore established for that purpose. Chapter 5 of the *MicroPower/Pascal Run-Time Services Manual* discusses shared memory usage.

The CREATE_SHARED_REGION request provides the complementary create operation, which declares a region as being shareable and assigns its run-time name.

Syntax

```
DELETE_SHARED_REGION ( { DESC := structure-desc }
                       { NAME := region-name    }
                       [[ STATUS := status-record ]] )
```

structure-descriptor
The identifier of a variable of predefined type STRUCTURE_DESC that contains the shared region's structure identifier.

region-name
A character-string constant or the identifier of a variable of predefined type NAME_STR that contains the 6-character name of an existing shared region (see Section 11.1.1.1).

status-record
The identifier of a variable of predefined record type EXC_STATUS that may receive an exception type and code. If you specify this parameter, the exception status, either success or error, that results from issuing the request is reported in this variable. Otherwise, an error causes the corresponding exception to be reported. The format of the exception record is described in Section 11.1.2.

Example

```
%INCLUDE 'DRAM.PAS'

TYPE
  Common_1 = ARRAY [1..%O'10000'] OF INTEGER;

VAR
  Rib_1 : REGION_ID_BLOCK;
  Reg_1 : ^Common_1;
```

```
[PRIORITY(10), STACK_SIZE(100)] PROCESS A;
BEGIN

  (* Delete a shared region. *)
   DISPOSE
     (Reg_1);
   DELETE_SHARED_REGION
     (NAME := 'REG1  ');

END; (* Process A *)
```

Semantics

The DELETE_SHARED_REGION procedure looks for a SRD (a kernel data structure) identified by the caller's structure descriptor block (SDB). If that SRD exists, the procedure deletes the SRD, removes the structure name from the system name table, and returns to the caller. If no such SRD exists, the procedure returns to the caller with an error indication.

This request is implemented through the DLST$ kernel primitive

Error Returns

See Section 11.2 for general information about error returns. The following exception codes may be returned:

ES$CDN	(type: SYSTEM_SERVICE)—Cannot specify both descriptor and name
ES$IST	(type: SYSTEM_SERVICE)—Invalid structure descriptor; shared region does not exist
ES$MDN	(type: SYSTEM_SERVICE)—Must specify descriptor or name

18.6 GET_MAPPING

MACRO equivalent: GMAP$

The GET_MAPPING procedure lets the calling process obtain a copy of its own current mapping or that of any other specified process. The request, valid only in mapped memory environments, returns the mapping information stored in the mapping-context restore area associated with the process control block (PCB) of the subject process to a record variable specified in the call.

The mapping information consists of a record of type MAPPING containing 16 fields of page address register (PAR) and page descriptor register (PDR) values. If instruction- and data-space (I&D-space) separation is in effect for the subject process, the record is of type ID_MAPPING and contains 32 fields of information: values for both the instruction and data APR sets.

Note

> Though separate I&D-space mapping is possible on an LSI–11/73 or similar target system, it may not necessarily be in effect for a given process.

Syntax

```
GET_MAPPING (  INFO := mapping-record
               ⟦ { DESC := process-desc  } ⟧
               ⟦ { NAME := process-name  } ⟧
               ⟦ STATUS := status-record ⟧ )
```

mapping-record

The identifier of a variable of predefined type MAPPING or ID_MAPPING that is to receive process mapping information.

process-descriptor

The identifier of a variable of predefined type PROCESS_DESC that contains the process identifier. The variable must have been previously initialized by an INIT_PROCESS_DESC request or by a process invocation statement with the DESC parameter (see Chapter 5).

process-name

Either a character-string constant or the identifier of a variable of predefined type NAME_STR that contains the 6-character name of an existing process (see Section 11.1.1.1).

status-record

The identifier of a variable of predefined record type EXC_STATUS that may receive an exception type and code. If you specify this parameter, the exception status, either success or error, that results from issuing the request is reported in this variable. Otherwise, an error causes the corresponding exception to be reported. The format of the exception record is described in Section 11.1.2.

If you specify neither a process-descriptor nor a process-name parameter, the procedure obtains the mapping status of the process issuing the request.

Format of Mapping Record

The information returned to the caller through the mapping-record parameter is a record of either type MAPPING or of type ID_MAPPING. Those types are declared in the system %INCLUDE file DRAM.PAS, as follows:

```
REGISTER_RANGE : ARRAY [0..7] OF UNSIGNED;

MAPPING = RECORD
            PARS : REGISTER_RANGE;
            PDRS : REGISTER_RANGE;
          END;

ID_MAPPING = RECORD
               I_PARS : REGISTER_RANGE;
               I_PDRS : REGISTER_RANGE;
               D_PARS : REGISTER_RANGE;
               D_PDRS : REGISTER_RANGE;
             END;
```

Type MAPPING is for target contexts in which separate I&D-space mapping is not in effect, such as an LSI-11/23.

PARS
: Contains the current values of the eight PARs. PARS[0] corresponds to PAR 0, and so on.

PDRS
: Contains the current values of the eight PDRs. PDRS[0] corresponds to PDR 0, and so on.

Type ID_MAPPING is for target contexts in which separate I&D-space mapping is in effect, such as provided by an LSI-11/73.

I_PARS
: Contains the current values of the eight I-space PARs. PARS[0] corresponds to I-space PAR 0, and so on.

I_PDRS
: Contains the current values of the eight I-space PDRs. PDRS[0] corresponds to I-space PDR 0, and so on.

D_PARS
: Contains the current values of the eight D-space PARs. PARS[0] corresponds to D-space PAR 0, and so on.

D_PDRS
: Contains the current values of the eight D-space PDRs. PDRS[0] corresponds to D-space PDR 0, and so on.

Note

The PDR word of any unused APR will contain 0. The content of the corresponding PAR is undefined and is not significant.

Example

```
%INCLUDE 'DRAM.PAS'

VAR
  Mapping_info : MAPPING;

[PRIORITY(10), STACK_SIZE(100), NAME ('A     ')] PROCESS A;
BEGIN

  (* Get the mapping information for this process. *)
  GET_MAPPING
    (INFO := Mapping_info,
     NAME := 'A     ');

END; (* Process A *)
```

Semantics

The GET_MAPPING procedure copies the contents of the mapping-context restore area specified by the process-descriptor or name parameters to the record variable specified in the call.

This request is implemented through the GMAP$ kernel primitive.

Error Returns

See Section 11.2 for general information about error returns. The following exception codes may be returned:

ES$CDN	(type: SYSTEM_SERVICE)—Cannot specify both descriptor and name
ES$IST	(type: SYSTEM_SERVICE)—Invalid structure description; process does not exist
ES$IPR	(type: SYSTEM_SERVICE)—Illegal primitive call; the request is illegal in an unmapped environment

The request may return the following error, though not as a result of standard Pascal programming practice:

ES$IAD	(type: SYSTEM_SERVICE)—Invalid address; pointer to buffer or structure is odd or not in user address space

Applications

Among other possible uses, the GET_MAPPING procedure lets a general-mapped process inspect its current mapping to identify unused APRs for use in dynamic mapping operations. This, in turn, lets the process optimize a sequence of mapping/remapping operations by using the FIXED mode of the call to MAP_WINDOW, which eliminates the need for intervening UNMAP_WINDOW calls.

18.7 MAP_WINDOW

MACRO equivalent: MAPW$

The MAP_WINDOW procedure, valid only in a mapped environment, permits a process to associate a sequence of virtual addresses with a specified region of physical memory. More precisely, MAP_WINDOW allows the calling process to extend or modify its virtual-to-physical mapping to include a previously unmapped area of physical memory. The caller supplies the physical description of a memory region, through a RIB, and specifies the portion of the region to be mapped. The MAP_WINDOW procedure then alters the calling process's MMU registers and PCB mapping context, normally by modifying one or more currently unused APRs, and returns an appropriate virtual address value to the caller. (Optionally, you can choose the APR or sequence of APRs to be modified.) Thus, the process obtains a virtual-address window into a region of memory that was not in its original address space.

The region may be a private physical region allocated to the process or may be a shared common or physical region previously accessed through the ACCESS_SHARED_REGION request.

The UNMAP_WINDOW request provides a complementary unmapping operation, which may be required between successive mapping operations, depending on the mode of MAP_WINDOW procedure usage. The main application objectives for the MAP_WINDOW and UNMAP_WINDOW procedures are the following:

- Usability by a general-mapped process, which cannot otherwise alter its mapping. (Other types of processes, which can perform direct MMU modification, may use MAP_WINDOW to alter mapping without the need for MMU-register saving during context switchouts, a performance consideration.)
- Use in conjunction with the ALLOCATE_REGION or ACCESS_SHARED_REGION requests, which provide the physical description of a memory region in the required format.

Therefore, the MAP_WINDOW procedure is described in terms of that primary application context. MAP_WINDOW and UNMAP_WINDOW can be used by processes with DEV_ACCESS, PRIVILEGED, or DRIVER mapping, of course, and also for mapping of objects other than memory regions as such.

Assuming a general-mapped process that does not borrow (force remapping of) an already allocated APR, the minimum requirement for using MAP_WINDOW is that the calling process's statically allocated virtual-address space not exceed 28K words when I&D-space separation is not in effect. In other words, at least one of the static process's APRs must remain unused, or inactive, at application build time. This requirement can be overridden by the fixed APR option, which forces MAP_WINDOW to use an APR indicated by the caller rather than the first unused APR that it finds. A dynamic process inherits the entire address space of its parent process and might not need all of that inherited mapping, as discussed in Chapter 5 of the *MicroPower/Pascal Run-Time Services Manual.*

The size of a window is controlled by a user-specified length parameter, which implies the number of APRs needed for the window. Thus, a process can map to an entire multipage region in a single operation, given that enough APRs are available for modification. If the caller does not have multiple APRs available for the window, and the region to be mapped is larger than 4K words (one virtual page), the process can step through the region by repeated mappings of a single APR, using suitably incremented window offsets. The potential size of a window is constrained only by the number of contiguous APRs available for the mapping, not by the size of the region as described in the RIB. To prevent "overmapping," you must ensure that the requested window length does not cause the window to extend beyond the end of the region.

The information supplied in the RIB variable that is specified in the call contains the region's location (physical base address and byte offset, if any), size, and mode attribute. The content of the RIB is assumedly that returned by a prior ALLOCATE_REGION or ACCESS_SHARED_REGION call; the format of the information is as described for those requests. In addition to the RIB, the caller supplies the length to map and an optional additive offset into the region specified in PAR ticks (32-word units); typically, a multiple of 128 ticks when stepping through a large region with a single-PAR window. The combination of those parameters determines the size and positioning of the mapped window within the region for a given call. The RIB content is never modified by MAP_WINDOW; the physical description of the region remains invariant throughout successive, incremental remappings. In general, you should not modify the RIB content.

Chapter 5 of the *MicroPower/Pascal Run-Time Services Manual* discusses region sharing and mapping, including the use of MAP_WINDOW in the context of other related requests. The ACCESS_SHARED_REGION and ALLOCATE_REGION requests provide the supporting operations that obtain RIB information. The GET_MAPPING, SAVE_CONTEXT, and RESTORE_CONTEXT requests provide additional support for mapping operations that involve borrowing of one or more APRs.

Syntax

```
MAP_WINDOW (  [[ ADDRESS_SPACE := { D_SPACE } ]]
                                  { I_SPACE }
              [[ ACCESS := { READ_WRITE } ]]
                           { READ_ONLY  }
              [[ PAR_CHOICE := { FREE  } ]]
                               { FIXED }
              [[ CACHING := { LEAVE   } ]]
                            { DISABLE }
              WINDOW_PTR := window-pointer
              [[ OFFSET := region-offset ]]
              LENGTH := window-length
              RIB := region-id-block
              [[ STATUS := status-record ]] )
```

ADDRESS_SPACE

Specifies whether the operation is to modify the process's D-space APR set. D_SPACE, the default, indicates yes; I_SPACE specifies that the operation is to modify the process's I-space APR set.

These parameters are meaningful only if I&D-space separation, possible in an LSI-11/73 target processor, is in effect for the calling process. Otherwise, the parameter is ignored.

ACCESS

Specifies that the operation map the window for read/write access (READ_WRITE, the default) or for read-only access (READ_ONLY).

PAR_CHOICE

Specifies the method of APR selection. FREE, the default, indicates that the operation modifies the free APR(s) chosen by MAP_WINDOW. (See Use of the Window-Pointer Parameter for further details.) FIXED indicates that the operation modifies the APR(s) that you specify by means of the window-pointer parameter.

CACHING

Specifies that the operation leave caching as is, either enabled or disabled (LEAVE, the default) or that the operation disable caching for this window, setting bit 15 of each PDR to disable caching for each APR (DISABLE). This operation is necessary on the arbiter side when you map to a KXJ shared memory area, if the arbiter uses cache memory; but even if this symbol is specified on an arbiter processor not using cache memory, there are no adverse effects. See Appendix B of the *MicroPower/Pascal I/O Services Manual* for additional information.

window-pointer

The identifier of a variable of type UNIVERSAL in which the request will return a virtual address corresponding to the first location in the mapped window, fully adjusted for offset(s), as described under Semantics. If the FREE option is specified, the precall value of this variable is not significant, and MAP_WINDOW chooses the APR(s) to use for mapping. If FIXED is specified, however, MAP_WINDOW uses the value you specify in this variable to select the first or only APR to be modified, as described under Use of the Window-Pointer Parameter.

region-offset

An unsigned integer value that is the desired displacement of the virtual window from the beginning of the region, expressed in PAR ticks (32-word units). This parameter is used when stepping through a large region by incremental remapping of a window. The default value is 0.

window-length

A constant or the identifier of a variable of type UNSIGNED that specifies size, in bytes, of the virtual window.

region-id-block

The identifier of a variable of predefined type REGION_ID_BLOCK that contains the location, size, and mode attribute of the region to map to, as described under Semantics.

status-record

The identifier of a variable of predefined record type EXC_STATUS that may receive an exception type and code. If you specify this parameter, the exception status, either success or error, that results from issuing the request is reported in this variable. Otherwise, an error causes the corresponding exception to be reported. The format of the exception record is described in Section 11.1.2.

Restrictions

- If I&D-space separation is in effect for the calling process, the combination of the I_SPACE option and the READ_WRITE (default) option is invalid.
- If you choose the FREE (default) APR selection option, calls to the UNMAP_WINDOW request are required between successive calls to MAP_WINDOW for iterative remapping of a window.
- The MAP_WINDOW (MAPW$) request may not be used to modify APR 0 of a process without I&D-space separation or D-space APR 0 of a process with I&D-space separation if that process accesses a supervisor-mode library.

Use of the Window-Pointer Parameter

In general, if you use the FREE (default) option, the variable specified by window-pointer is a destination-only variable, but if you select the FIXED option, window-pointer is both a source and destination variable.

More specifically, if you select the FREE option, whether explicitly or by default, MAP_WINDOW ignores the content of the window-pointer variable and selects one or more free APRs for the mapping operation. MAP_WINDOW returns the virtual address corresponding to the first or only APR selected for the window in the window-pointer variable.

If you select the FIXED option, however, the variable specified by window-pointer prior to the call must contain a virtual address in the range of the first or only APR to be modified by the operation. Thus, you can force the selection of APRs when you use the FIXED option.

For example, if the precall value in window-pointer is 140000(octal), corresponding to the base of APR 6, the request uses APR 6 and, if needed, APR 7 for the mapping operation, regardless of the free or in-use status of those APRs. For the purposes of this example, the value in window-pointer could be any address within the virtual page beginning at 140000 (that is, could be from 140000 to 157776) with exactly the same effect.

The virtual address value returned in window-pointer would be 140000 plus any common-region offset contained in the RIB for the region in question. Normally, the returned address would be exactly 140000 for a physical region or a value between 140000 and 140076 for a shared common region.

If you use the FIXED APR selection option, calls to UNMAP_WINDOW are not required between successive calls to MAP_WINDOW for iterative remapping of a window.

Example

```
%INCLUDE 'DRAM.PAS'

VAR
  Rib_1 : REGION_ID_BLOCK;
  W_ptr : UNIVERSAL;

[PRIORITY(10), STACK_SIZE(100)] PROCESS A;
BEGIN
```

```
  (* Map the window. *)
  MAP_WINDOW
    (PAR_CHOICE := FIXED,
     WINDOW_PTR := W_ptr,
     OFFSET := 0,
     LENGTH := %O'20000',
     RIB := Rib_1);

END; (* Process A *)
```

Semantics

In the following description, free APR refers to an APR that is unmapped (whose access control field is set to no access) at the time of the call. Only free APRs are candidates for modification under the FREE option. (An APR that was modified by a prior call to MAP_WINDOW can be freed for remapping by an intervening call to UNMAP_WINDOW.)

The MAP_WINDOW procedure calculates the number of APRs, *n*, needed for the window, based on the window-length value specified in the call plus the region offset, if any, described in the RIB. If I&D-space separation is in effect for the calling process, MAP_WINDOW selects the APR(s) to be operated on as specified by the D_SPACE or I_SPACE option.

If the FIXED option was specified, MAP_WINDOW determines the initial or only APR to be mapped, APRi, from the virtual address value supplied in the window-pointer variable. If more than one APR is needed and *n* APRs do not exist starting with APRi, MAP_WINDOW returns to the caller, with an error indicating "too few APRs available." If the FREE option was specified explicitly or by default, MAP_WINDOW tests the caller's mapping context for *n* consecutive free APRs. If *n* consecutive free APRs are not available, MAP_WINDOW returns to the caller with an error indicating "too few APRs available." Otherwise, the first of the *n* free APRs is established as APRi.

MAP_WINDOW then maps the required APRs, modifying both the MMU hardware registers and the corresponding locations in the mapping-context restore area associated with the caller's PCB. MAP_WINDOW forms the physical base address, or PAR value, for APRi by adding the offset specified in the call (in PAR ticks) to the region base described in the RIB. PAR values for successive APRs, if any, are incremented appropriately. Page descriptor register (PDR) values, specifying access control and page lengths, are set as required.

Finally, MAP_WINDOW forms the window-pointer address by adding the region offset, if any, described in the RIB to the 4K-boundary virtual address that corresponds to APRi, and returns that value to the window-pointer variable specified in the call.

Information describing the region to be mapped is supplied in a variable specified by the region-id-block parameter. The variable is of the form defined in the DRAM.PAS %INCLUDE file.

```
                                         {Mode}
                                  {Physical    Common}
REGION_ID_BLOCK = RECORD
  REGION_ADDRESS : UNIVERSAL;     {PAR value / PAR value}
  REGION_SIZE : UNSIGNED;         {        ignored       }
  REGION_MODE : ADDRESS_TYPE;     {PHYSICAL   / COMMON   }
  REGION_OFFSET : UNSIGNED;       {0          / bytes    }
  END;
```

REGION_ADDRESS

An unsigned integer that specifies the region's base address and must be a physical PAR value (always on a 32-word boundary).

REGION_SIZE

The number of PAR ticks (units of 32 words) contained in the region. This parameter is not used in the MAP_WINDOW operation, since the window-length parameter in the call determines the length of the mapped window.

REGION_MODE

The enumerated type values COMMON and PHYSICAL, denoting a common or physical region, respectively. This field is not checked by MAP_WINDOW.

REGION_OFFSET

Relevant only for a shared common region, an increment, in bytes, from the region-base PAR value to the beginning of the region. The ALLOCATE_REGION, ACCESS_SHARED_REGION, and CREATE_SHARED_REGION requests will supply an appropriate value for this field.

This request is implemented through the MAPW$ kernel primitive.

Error Returns

See Section 11.2 for general information about error returns. The following exception codes may be returned:

ES$IPR	(type: SYSTEM_SERVICE)—Illegal primitive call; the request was issued in an unmapped environment
ES$NFA	(type: RESOURCE)—No free APR; insufficient number of APRs available for the requested operation (see Semantics)

The request may return the following error, though not as a result of standard Pascal programming practice:

ES$IAD	(type: SYSTEM_SERVICE)—Invalid address; the RIB address is not on a word boundary

Implementation Notes

Since the MMU-register modifications MAP_WINDOW and UNMAP_WINDOW perform are reflected by corresponding changes in the caller's mapping-context restore area in one logically indivisible operation, MMU-context saving is not required each time the process is switched out of the run state. Such context saving is needed by processes that modify their mapping directly by accessing the I/O page, at some cost in overall performance. (MMU-context saving is a process creation option.) This aspect of MAP_WINDOW usage versus self-modification should be weighed in the design of DRIVER, PRIVILEGED, and DEVICE_ACCESS processes that require dynamic mapping alterations. The SAVE_CONTEXT and RESTORE_CONTEXT requests facilitate saving and restoring of initial mapping values when using the MAP_WINDOW and UNMAP_WINDOW requests.

18.8 RESTORE_CONTEXT

MACRO equivalent: RCTX$

The RESTORE_CONTEXT procedure permits a process to reset itself to an earlier state of virtual-to-physical mapping previously saved using the SAVE_CONTEXT procedure. RESTORE_CONTEXT restores the APR values that were most recently saved by SAVE_CONTEXT and updates the mapping-context restore area associated with the caller's PCB accordingly. (The mapping-context restore area contains the process's mapping image and is used automatically by the kernel during process context switches.)

Used in conjunction with the SAVE_CONTEXT procedure, RESTORE_CONTEXT allows a process to reset its entire mapping to a known state, canceling the effect of intervening alterations of its mapping, especially if such mapping operations involved borrowing of one or more statically mapped APRs.

Multiple calls to RESTORE_CONTEXT without intervening calls to SAVE_CONTEXT cause successively older generations of mapping context to be restored, assuming that multiple save operations were executed prior to the sequence of calls to RESTORE_CONTEXT. Multiple copies of mapping context are saved in LIFO order, as described for the SAVE_CONTEXT procedure. Thus, a process could take snapshots of its mapping at several points and then restore the last-saved mapping, the next-to-last, and so on, by a corresponding number of calls to RESTORE_CONTEXT.

Syntax

RESTORE_CONTEXT

Example

```
%INCLUDE 'DRAM.PAS'

[PRIORITY(10), STACK_SIZE(100)] PROCESS A;
BEGIN

  (* Restore the mapping context. *)
  RESTORE_CONTEXT;

END; (* Process A *)
```

Semantics

The RESTORE_CONTEXT procedure copies the mapping-register image contained in the first or only context descriptor block pointed to by the caller's PCB into both the MMU registers and the mapping-context restore area used for process context switching. The procedure then removes the block from the caller's context-descriptor list, deallocates the block, and returns to the caller.

If the caller's context-descriptor list is empty, the procedure returns an error indication.

This request is implemented through the RCTX$ kernel primitive.

Error Returns

See Section 11.2 for general information about error returns. The following exception code may be returned:

ES$IPR	(type: SYSTEM_SERVICE)—Illegal primitive call; no mapping context has been saved; SAVE_CONTEXT or RESTORE_CONTEXT was issued in an unmapped environment

Implementation Notes

Like MAP_WINDOW and UNMAP_WINDOW, the RESTORE_CONTEXT procedure alters both the MMU hardware registers and the caller's automatic mapping-context restore area in one logically indivisible operation. Thus, if all mapping alterations are done exclusively through MAP_WINDOW and RESTORE_CONTEXT operations, MMU-context saving is not required each time the process is switched out of run state. Such context saving is needed by a process that modifies its mapping directly by accessing to the I/O page, at some cost in overall performance. (MMU-context saving is a process creation option.) This aspect of MAP_WINDOW/RESTORE_CONTEXT usage versus self-modification should be weighed in the design of DRIVER, PRIVILEGED, and DEVICE-ACCESS processes that require dynamic mapping alterations.

18.9 SAVE_CONTEXT

MACRO equivalent: SCTX$

The SAVE_CONTEXT procedure permits a process to save a copy of its memory mapping for subsequent restoration by using the RESTORE_CONTEXT procedure. SAVE_CONTEXT saves the contents of the calling process's mapping registers (APRs) in a context block that is distinct from the mapping-context restore area always associated with a process's PCB. (The latter area is used implicitly by the kernel during process context switching.)

Used with the RESTORE_CONTEXT procedure, SAVE_CONTEXT allows a process to reset its entire mapping to a prior, known state, canceling the effect of intervening alterations of its mapping, especially if such mapping operations involved borrowing of one or more statically mapped APRs. Typically, SAVE_CONTEXT might be used preceding a fixed-mode call to MAP_WINDOW.

Assuming that the remapping is of a temporary nature, a call to RESTORE_CONTEXT would be used at some later point to restore the previous mapping.

Successive calls to SAVE_CONTEXT without intervening calls to RESTORE_CONTEXT cause multiple copies of mapping context to be saved in a list structure treated by RESTORE_CONTEXT as a LIFO push-down stack. Thus, a process could take snapshots of its mapping at various points and then restore the last-saved mapping, the next-to-last, and so on, by an appropriate number of successive calls to RESTORE_CONTEXT.

Together, the SAVE_CONTEXT and RESTORE_CONTEXT procedures facilitate easy, uncomplicated restoration of mapping, at a relatively small cost in performance. Also, if used with MAP_WINDOW (as opposed to direct MMU modification) that set of requests eliminates the need for MMU-register saving during process context switches from run state, an overall performance benefit.

Chapter 5 of the *MicroPower/Pascal Run-Time Services Manual* discusses dynamic mapping.

Syntax

SAVE_CONTEXT (⟦ STATUS := status-record ⟧)

status-record

The identifier of a variable of predefined record type EXC_STATUS that may receive an exception type and code. If you specify this parameter, the exception status, either success or error, that results from issuing the request is reported in this variable. Otherwise, an error causes the corresponding exception to be reported. The format of the exception record is described in Section 11.1.2.

Example

```
%INCLUDE 'DRAM.PAS'

[PRIORITY(10), STACK_SIZE(100)] PROCESS A;
BEGIN

  (* Save the mapping context. *)
  SAVE_CONTEXT;

END; (* Process A *)
```

Semantics

The SAVE_CONTEXT procedure allocates a context descriptor block in system-common memory and copies the contents of the user's MMU registers into the block. The procedure links the block into the context-descriptor list pointed to by the caller's PCB, as the first or only element of that list, and returns to the caller.

In an LSI–11/73 or similar target environment, the procedure saves both the I&D-space mapping registers if I&D-space separation is in effect for the calling process. Otherwise, only the I-space APR set is saved.

This request is implemented through the SCTX$ kernel primitive.

Error Returns

See Section 11.2 for general information about error returns. The following exception codes may be returned:

ES$IPR	(type: SYSTEM_SERVICE)—Illegal primitive call; SAVE_CONTEXT was issued in an unmapped environment
ES$NMK	(type: RESOURCE)—Insufficient space for kernel structure; a context descriptor block could not be allocated

18.10 UNMAP_WINDOW

MACRO equivalent: UMAP$

The UNMAP_WINDOW procedure permits a process to reverse the effect of a prior MAP_WINDOW operation, dissociating a sequence of virtual addresses (the virtual window) from the physical memory to which it was mapped. (The procedure is valid only in a mapped environment.) More precisely, UNMAP_WINDOW sets the APR(s) corresponding to a specified window to inactive or no access and modifies the calling process's mapping context to reflect the availability of the APR(s) for subsequent remapping.

The caller identifies the window to be unmapped by supplying the base virtual address of the window and a length to unmap. The address is presumably one previously returned by the MAP_WINDOW procedure. The MAP_WINDOW request provides the complementary window-mapping operation. An explicit unmapping operation is required between successive mapping operations that remap a given window in free mode. However, if the fixed mode of MAP_WINDOW operation is used for the remapping, intervening UNMAP_WINDOW calls are unnecessary.

Chapter 5 of the *MicroPower/Pascal Run-Time Services Manual* discusses dynamic mapping, including the use of UNMAP_WINDOW in the context of other related requests. The SAVE_CONTEXT and RESTORE_CONTEXT requests provide additional support for mapping operations that involve borrowing of one or more APRs.

Syntax

```
UNMAP_WINDOW ( [[ ADDRESS_SPACE := { D_SPACE } ]]
                                  { I_SPACE }
               LENGTH := window-length
               WINDOW_PTR := window-pointer )
```

ADDRESS_SPACE

Specifies that the operation is to modify the process's D-space (D_SPACE, default) APR set or the process's I-space (I_SPACE) APR set.

These parameters are meaningful only if I&D-space separation, possible in an LSI–11/73 target processor, is in effect for the calling process. Otherwise, the parameter is ignored.

window-length

A constant or the identifier of a variable of type UNSIGNED that specifies the size, in bytes, of the virtual window to be unmapped. This value effectively determines the number of APRs that are unmapped, or freed, by the operation.

window-pointer

The identifier of a variable of type UNIVERSAL that contains the virtual address that identifies the window to be unmapped. Normally, this value is supplied by a prior call to MAP_WINDOW.

Example

```
%INCLUDE 'DRAM.PAS'

VAR
  W_ptr : UNIVERSAL;

[PRIORITY(10), STACK_SIZE(100)] PROCESS A;
BEGIN

  (* Unmap the window. *)
  UNMAP_WINDOW
    (WINDOW_PTR := W_ptr,
     LENGTH := %O'20000');

END; (* Process A *)
```

Semantics

The UNMAP_WINDOW procedure determines which APR(s) map the window identified in the request and clears the corresponding PDR(s), effectively setting the access control field of the affected APR(s) to no access. The unmapping operation is performed on both the MMU hardware registers and the corresponding locations in the mapping-context restore area associated with the caller's PCB.

This request is implemented through the UMAP$ kernel primitive.

Error Returns

See Section 11.2 for general information about error returns. The following exception code may be returned:

ES$IPR	(type: SYSTEM_SERVICE)—Illegal primitive call; UNMAP_WINDOW was issued in an unmapped environment

Chapter 19
Clock Service Requests

This chapter describes the requests that provide for setting and obtaining the kernel-maintained system time and for timed process blocking. These requests, implemented through the predeclared procedures listed in Table 19–1, are the Pascal language interface to the services provided by the kernel's timer primitives.

Note

To use these requests, a system clock must be present and configured on your target system.

You must include in your program or module the definitions of the routines that implement the requests described in this chapter before using them. See Appendix I for more information.

Table 19–1 summarizes the Pascal clock service requests.

Table 19–1: Clock Service Requests

Request	Operation
GET_TIME	Obtains the system time
SET_TIME	Sets the system time to an arbitrary value
SLEEP	Blocks the calling process in the wait-active state until a specified time interval has elapsed

19.1 About System Time

The kernel calculates system time in milliseconds on the basis of interrupts generated by a 50 Hz, 60 Hz, 100 Hz, or 800 Hz clock source. Thus, the time is kept in multimillisecond granules or clock ticks. (For example, a 60 Hz clock ticks only once every 16.7 milliseconds.) Chapter 3 of the *MicroPower/Pascal Run-Time Services Manual* describes the possible range of discrepancy between reported system time and actual elapsed time due to clock frequency and discusses the further effect that relative process priorities may have on reported time as perceived by the calling process. Chapter 3 of the *MicroPower/Pascal Run-Time Services Manual* also provides additional information about the kernel's primitive clock services.

19.2 Format of CLOCK_TIME Record

The CLOCK_TIME record facilitates manipulation of the triple-precision (48-bit) system time value used by clock service requests and the auxiliary routines.

```
CLOCK_TIME = RECORD
  CASE Clock_time_use OF

    (* Used by GET_TIME and SET_TIME *)
       Clock_value : (Low, Middle, High : UNSIGNED);

    (* Used by SLEEP *)
       Sleep_interval : (Interval : LONG_INTEGER; Overflow : UNSIGNED);
  END;
```

Low
An unsigned value that is the low-order 16 bits of time value.

Middle
An unsigned value that is the middle-order 16 bits of time value.

High
An unsigned value that is the high-order 16 bits of time value.

Interval
A long-integer value that is the low-order 32 bits of the 48-bit time value. (This value is used by the SLEEP request.)

Overflow
An unsigned value that is the high-order 16 bits of the 48-bit time value. (This overflow count is for SLEEP request values that exceed the magnitude of the SLEEP request's parameter.)

19.3 GET_TIME

MACRO equivalent: GTIM$

The GET_TIME procedure obtains the approximate system time. System time is either of the following:

1. The elapsed time since the last system initialization (zero based).
2. The base time set by the SET_TIME procedure plus the elapsed time since the base system time was last set. (A base time, if used, is normally set as part of the system startup or restart procedures.)

GET_TIME obtains the system time in milliseconds as a 48-bit triple-precision value. GET_TIME returns this value in a record variable that contains three unsigned integer fields: a low-order value field, a middle-order value field, and a high-order value field. This value allows for an extremely large maximum elapsed time—more than 4400 years, assuming a zero base. The calling process may need to consider only the low-order or low- and middle-order portions of the time value, as discussed in Chapter 3 of the *MicroPower/Pascal Run-Time Services Manual.*

The SLEEP procedure provides a related process blocking-and-wakeup service based on elapsed system time.

Syntax

GET_TIME (SYS_TIME := time-record)

time-record

The identifier of a variable of predefined record type CLOCK_TIME that will receive the system time value. See Section 19.2.

Example

```
%INCLUDE 'TIMER.PAS'

VAR
  T : CLOCK_TIME;

PROCESS Stop_watch;
BEGIN

  (* Get the system time. *)
  GET_TIME
    (SYS_TIME := T);

END; (* Process Stop_watch *)
```

Semantics

The GET_TIME procedure copies the 48-bit system time value into three unsigned values that comprise the variable specified in the call and returns to the caller.

This request is implemented through the GTIM$ kernel primitive.

Error Returns

The request may return the following error, though not as a result of standard Pascal programming practice:

ES$IAD	(type: SYSTEM_SERVICE)—Invalid address; odd or not in user space

19.4 SET_TIME

MACRO equivalent: STIM$

The SET_TIME procedure sets the system time maintained by the kernel to an arbitrary base time value, assuming that a system clock is present and configured on the target system.

SET_TIME obtains the new system time in milliseconds from a 48-bit triple-precision value specified in the call. This record variable contains three unsigned integer fields: a low-order value field, a middle-order value field, and a high-order value field. The 48-bit value allows for an extremely large maximum elapsed time—more than 4400 years, assuming a zero base. Chapter 3 of the *MicroPower/Pascal Run-Time Services Manual* discusses the clock and timer services provided by the kernel.

Syntax

SET_TIME (SYS_TIME := time-record)

time-record

The identifier of a variable of predefined record type CLOCK_TIME that will receive the system time value. See Section 19.2.

Example

```
%INCLUDE 'TIMER.PAS'

VAR
  T : CLOCK_TIME;

PROCESS Stop_watch;
BEGIN

  (* Set the system time. *)
  WITH T DO
   BEGIN
     LOW := 0;
     MIDDLE := 0;
     HIGH := 0;
   END;
  SET_TIME
    (SYS_TIME := T);

END; (* Process Stop_watch *)
```

Semantics

The SET_TIME procedure copies the 48-bit time value specified by the three unsigned values of the variable specified in the call into the kernel's system time variable and returns to the caller.

This request is implemented through the STIM$ kernel primitive.

Error Returns

The request may return the following error, though not as a result of standard Pascal programming practice:

ES$IAD (type: SYSTEM_SERVICE)—Invalid address; odd or not in user space

19.5 SLEEP

MACRO equivalent: SLEP$

The SLEEP procedure blocks the calling process in the wait-active state until the closest approximation in clock ticks equal to or exceeding the specified time interval has elapsed. At that point, the process changes to the ready-active state, from which it may be switched to the run state, depending on the relative priorities of the unblocked process and the current running process. (If the process was suspended during the blocking interval, the process changes to ready suspended rather than ready active, of course.)

The caller specifies the blocking interval as the number of milliseconds following execution of the call to SLEEP. The blocked process is never unblocked in less than the specified time. The range of positive difference between the specified and actual blocking time is a function of both the clock frequency and relative process priorities. (A 60 Hz system clock, for example, ticks only once every 16.7 milliseconds.) Chapter 3 of the *MicroPower/Pascal Run-Time Services Manual* describes the range of possible differences and discusses techniques for eliminating or minimizing any discrepancy.

The blocking, specified as a positive long-integer value, can range from one millisecond (useful only with an 800 Hz clock) to roughly 24.9 days.

Use of SLEEP assumes that a system clock is present and configured on the target system.

Syntax

SLEEP (INTERVAL := blocking-interval)

blocking-interval

A constant or the identifier of a variable of predefined type LONG_INTEGER that specifies the time, in milliseconds, that is the desired interval during which the process is blocked. The value must be from 0 to (2**31) -1.

Example

```
%INCLUDE 'TIMER.PAS'

PROCESS Stop_watch;
BEGIN

  (* Go to sleep for a while. *)
  SLEEP
    (INTERVAL := 100000);

END; (* Process Stop_watch *)
```

Semantics

The SLEEP procedure blocks the calling process on the system timer queue, adjusting the queue order and current expiration values as required, and calls the scheduler. The queue is time-ordered; blocked processes are queued on it in ascending order of blocking interval time values.

If the blocking interval value supplied in the call is 0, SLEEP treats the request as a null operation and returns control to the caller.

At each tick of the system clock, the kernel's clock interrupt service updates the system time, checks the timer queue, and unblocks any process(es) whose blocking interval has expired. Each unblocking implies a possible scheduler call.

This request is implemented through the SLEP$ kernel primitive.

Restriction

The interval value is limited to a 31-bit positive integer; that is, the sign bit of the high-order word must not be set.

Error Returns

See Section 11.2 for general information about error returns. The following exception code may be returned:

ES$IPM	(type: SYSTEM_SERVICE)—Illegal parameter; a negative interval was specified (value exceeds (2**31) –1)

The request may return the following error, though not as a result of standard Pascal programming practice:

ES$IAD	(type: SYSTEM_SERVICE)—Invalid address; odd or not in user space

Chapter 20
Miscellaneous Requests

This chapter describes a miscellaneous group of requests. Table 20–1 lists the predeclared procedures and functions that are the Pascal language interface to primitive services provided by the kernel.

You must include in your program or module the definitions of the routines that implement the requests described in this chapter before using them. See Appendix I for more information.

Table 20–1: Miscellaneous Requests

Request	Operation
CHECK_FREE_SPACE	Obtains heap storage space information
CREATE_LOGICAL_NAME	Defines or redefines a 1- to 6-character logical name
DELETE_LOGICAL_NAME	Eliminates the translation value defined for a given logical name, effectively "undefining" the name
GET_CONFIG	Obtains hardware configuration information
POWER_FAIL	Detects whether a recovery from a power failure is in progress
TRANSLATE_LOGICAL_NAME	Obtains the translation string defined for a given logical name

20.1 About Logical Names

A logical name is a kernel structure that contains the translation value for the name. Logical names provide a way of establishing equivalent values for strings, thus allowing you to write generic applications that will execute differently, depending on the current value of a logical name. For example, by associating a file specification with a logical name, an application could communicate with a serial line, a file on disk, or a communication port, with no change to the source code. Thus, a program could reference a default device named "DK:", and the file system will have the logical name DK assigned to a real device, say, DYA0:.

You can also use logical names to represent kernel resources such as semaphores, ring buffers, or process run-time names for later translation by the other real-time programming requests. The real-time programming requests (other than the logical name requests) and I/O requests that use file names will automatically translate a logical name, provided as a name parameter, to its equivalence string when the request is invoked. The translation will traverse multiple levels of serial logical name definitions to the final equivalence string. Refer to Chapter 3 of the *MicroPower/Pascal Run-Time Services Manual* for additional details about logical names.

20.2 CHECK_FREE_SPACE

MACRO equivalent: none

The CHECK_FREE_SPACE procedure obtains information on heap storage space. The procedure returns the total size of free space and the largest contiguous segment of free space in heap storage.

Syntax

```
CHECK_FREE_SPACE ( TOTAL := total-free-space
                   LARGEST := largest-contiguous-segment )
```

total-free-space

The identifier of an unsigned variable that receives the size in bytes of the free heap storage space.

largest-contiguous-segment

The identifier of an unsigned variable that receives the size in bytes of the largest contiguous segment of free heap storage space.

Example

```
%INCLUDE 'MISC.PAS'

VAR
  Total, Largest : UNSIGNED;

PROCEDURE Free_Space;
BEGIN

  CHECK_FREE_SPACE(Total, Largest);
  WRITELN('Total: ',Total);
  WRITELN('Largest: ',Largest);

END;
```

Semantics

The CHECK_FREE_SPACE procedure walks through the linked list of free heap storage space to calculate the total free space. The size of each free segment is added to a running total to calculate TOTAL. The largest segment size (LARGEST) is determined by comparing each segment size to the previous largest size and setting LARGEST equal to the current segment size when it is larger than the previous largest size.

Error Returns

None

20.3 CREATE_LOGICAL_NAME

MACRO equivalent: CRLN$

The CREATE_LOGICAL_NAME procedure allows the caller to define or redefine a logical name and associate it with a translation string. More precisely, the procedure creates a logical name data structure containing a user-specified translation string value for a given name. Subsequent instances of the logical name will be automatically translated to the corresponding value by the other real-time requests that operate on dynamic data structures, described in Chapters 12 through 19. An override option permits a preexisting logical name definition to be replaced, thus redefining the name.

The complementary TRANSLATE_LOGICAL_NAME request returns the translation string value directly associated with a logical name, and the DELETE_LOGICAL_NAME request eliminates the translation-string value associated with a currently defined logical name.

Syntax

```
CREATE_LOGICAL_NAME (  [[ OVERRIDE := { TRUE  } ]]
                                      { FALSE }
                       LENGTH : translation-string-length
                       STRING := translation-string
                       [[ DESC := logical-name-descriptor ]]
                       NAME := logical-name
                       [[ STATUS := status-record ]] )
```

OVERRIDE

If TRUE, the logical name already exists, and the character string provided by the translation-string parameter should replace the translation string associated with the logical name. FALSE, the default, specifies that a preexisting logical name definition will not be replaced.

translation-string-length

The identifier of a variable of predefined type LOGICAL_NAME_LEN or an integer constant that is the length, in bytes, of the character string specified by the translation-string parameter. The valid range of this parameter is 1 to 256

translation-string

A string constant or the identifier of a variable that contains a 1- to 256-character ASCII character string to be used as the translation string. (The effective length of the string is determined by the translation-string-length parameter.)

logical-name-descriptor

The identifier of a variable of predefined type STRUCTURE_DESC that is to receive the logical name's structure identifier.

logical-name

A character-string constant or the identifier of a variable of predefined type NAME_STR that specifies the 6-character name of the logical name structure (see Section 11.1.1.1).

status-record

The identifier of a variable of predefined record type EXC_STATUS that may receive an exception type and code. If you specify this parameter, the exception status, either success or error, that results from issuing the request is reported in this variable. Otherwise, an error causes the corresponding exception to be reported. The format of the exception record is described in Section 11.1.2.

Restrictions

- Like other structure names, a logical name must be unique among all names of kernel data structures.
- By system convention, if the translation value of a given logical name is itself intended as a logical name (through serial definitions) and the translation value consists of fewer than six printing characters, the name should be padded to six characters with trailing ASCII spaces in the supplied translation string.

Example

```
PROGRAM logical;
%INCLUDE 'LOGNAM.PAS'

VAR
  F1 : TEXT;
  V1 : STRUCTURE_DESC;

[INITIALIZE] PROCEDURE Init;
BEGIN

  (* Create a logical name. *)
  CREATE_LOGICAL_NAME (DESC := V1,
                       NAME := 'MY_DEV',
                       STRING := 'DYA0',
                       LENGTH := 4);

END; (* Procedure Init *)

BEGIN
  OPEN (F1, 'MY_DEV:TEST.DAT', history:=NEW);
END.
```

Semantics

The CREATE_LOGICAL_NAME procedure attempts to create a logical-name kernel data structure named as specified in the logical-name parameter and large enough to contain the supplied translation string. If successful, the procedure copies the translation string into the named structure and returns to the caller.

If the specified structure name is defined as a logical name and OVERRIDE is specified as TRUE, the procedure deletes the existing logical-name structure and attempts to create and fill in a new one. If the TRUE option was not specified or the structure name is in use as other than a logical, the procedure returns to the caller, with a "name already in use" error indication.

If the structure creation fails for some other reason, the procedure returns to the caller, with an appropriate error indication.

This request is implemented through the CRLN$ kernel primitive.

Error Returns

See Section 11.2 for general information about error returns. The following exception codes may be returned:

ES$NMK	(type: RESOURCE)—Insufficient space for kernel structure; the required logical-name structure could not be allocated
ES$MDN	(type: SYSTEM_SERVICE)—Must specify structure descriptor or name
ES$SNI	(type: SYSTEM_SERVICE)—Structure name in use; the name to be defined as a logical name conflicts with an existing structure name

The request may also return the following error, though not as a result of standard Pascal programming practice:

ES$IPM	(type: SYSTEM_SERVICE)—Illegal parameter; the specified string length exceeds 256

20.4 DELETE_LOGICAL_NAME

MACRO equivalent: DLLN$

The DELETE_LOGICAL_NAME procedure allows the caller to eliminate the translation value defined for a given logical name, effectively "undefining" the name. More precisely, DELETE_LOGICAL_NAME deletes the kernel data structure containing the translation string immediately associated with the name supplied in the call. (Contrast with the DESTROY procedure, which attempts to translate any logical name into the name of another type of structure and will not delete a logical-name structure. DELETE_LOGICAL_NAME, on the other hand, requires that the named structure be a logical-name value and will not perform any translation.)

The complementary CREATE_LOGICAL_NAME procedure defines the translation value associated with a logical name, and the TRANSLATE_LOGICAL_NAME procedure returns the translation value associated with a currently defined logical name.

Syntax

```
DELETE_LOGICAL_NAME ( { DESC := logical-name-descriptor }
                      { NAME := logical-name            }
                     [[ STATUS := status-record ]] )
```

logical-name-descriptor

The identifier of a variable of predefined type STRUCTURE_DESC that contains the structure identifier of the logical name to be deleted.

logical-name

A character string constant or the identifier of a variable of predefined type NAME_STR that specifies the 6-character name of the logical-name structure to be deleted (see Section 11.1.1.1).

status-record

The identifier of a variable of predefined record type EXC_STATUS that may receive an exception type and code. If you specify this parameter, the exception status, either success or error, that results from issuing the request is reported in this variable. Otherwise, an error causes the corresponding exception to be reported. The format of the exception record is described in Section 11.1.2.

Caution

Logical names are, by nature, often dynamically redefined. Since each redefinition can cause the creation of a new logical-name structure, a reference to the logical-name structure, using a descriptor (DESC parameter), may access an obsolete structure.

DIGITAL recommends as a safe programming practice that you refer to logical-name structures by name (NAME parameter) rather than by descriptor to avoid an accidental reference to an obsolete logical-name structure.

Example

```
%INCLUDE 'LOGNAM.PAS'

VAR
  Temp_file : STRUCTURE_DESC;

[TERMINATE] PROCEDURE Term;
BEGIN

  (* Delete a logical name. *)
  DELETE_LOGICAL_NAME (DESC := Temp_file);

END; (* Procedure Term *)
```

Semantics

The DELETE_LOGICAL_NAME procedure verifies that the kernel data structure identified in the call is a logical-name structure and, if so, deletes the structure and removes the corresponding name from the system name table.

This request is implemented through the DLLN$ kernel primitive.

Error Returns

See Section 11.2 for general information about error returns. The following exception codes may be returned:

ES$CDN	(type: SYSTEM_SERVICE)—Cannot specify both descriptor and name
ES$IST	(type: SYSTEM_SERVICE)—Invalid structure descriptor; no such logical-name structure exists
ES$MDN	(type: SYSTEM_SERVICE)—Must specify descriptor or name

20.5 GET_CONFIG

MACRO equivalent: GVAL$

The GET_CONFIG procedure obtains information about the target hardware configuration. The information returned is that supplied in the configuration file at application build time and does not necessarily reflect the hardware configuration of a particular target.

Syntax

```
GET_CONFIG ( CLOCK_FREQ := clock-frequency
             CONFIG := configuration-record
             [[ STATUS := status-record ]] )
```

clock-frequency

The identifier of an integer variable that will receive the frequency, in Hertz, of the real-time clock. The value received will be one of the following: 50, 60, 100, or 800 or 0 if there is no clock.

configuration-record

The identifier of a variable of predefined record type HARDWARE_CONFIG that will receive configuration information (see the Configuration Record Format).

status-record

The identifier of a variable of predefined record type EXC_STATUS that may receive an exception type and code. If you specify this parameter, the exception status, either success or error, that results from issuing the request is reported in this variable. Otherwise, an error causes the corresponding exception to be reported. The format of the exception record is described in Section 11.1.2.

Configuration Record Format

The configuration record is a variable of predefined type HARDWARE_CONFIG, as follows:

```
HARDWARE_CONFIG = PACKED RECORD
                    FPP : [POS(0)] BOOLEAN;
                    FIS : [POS(1)] BOOLEAN;
                    F11 : [POS(2)] BOOLEAN;
                    J11 : [POS(3)] BOOLEAN;
                    T11 : [POS(4)] BOOLEAN;
                    IOP : [POS(5)] BOOLEAN;
                    Q22 : [POS(6)] BOOLEAN;
                    MMU : [POS(7)] BOOLEAN;
                    CMR : [POS(8)] BOOLEAN;
                    FPA : [POS(9)] BOOLEAN;
                    ROM_RAM : [POS(15)] BOOLEAN;
                  END;
```

FPP

A Boolean value that, when TRUE, indicates that the target processor is configured with a KEF11 floating-point processor option.

FIS

A Boolean value that, when TRUE, indicates that the target processor is configured with a KEV11 floating-point option.

F11

A Boolean value that, when TRUE, indicates that the target processor is a KDF–11 or KDF–11B (contains the KTF11 CPU).

J11

A Boolean value that, when TRUE, indicates that the target processor is a KDJ11 (contains the DCJ11 CPU); or, if the IOP Boolean value is also TRUE, indicates that the target processor is a KXJ11–CA.

T11

A Boolean value that, when TRUE, indicates that the target processor is a KD11–F, KD11–HA, KXT–11A, or KXT11–CA (contains the T11 CPU).

IOP

A Boolean value that, when TRUE, indicates that the target processor is a KXT11–CA; or, if the J11 Boolean value is also TRUE, indicates that the target processor is a KXJ11–CA.

Q22

A Boolean value that, when TRUE, indicates that the system has 22-bit addressing capability.

MMU

A Boolean value that, when TRUE, indicates that the target processor is configured with a KT–11 memory-management unit.

CMR

A Boolean value that, when TRUE, indicates that the target processor is a CMR21.

FPA

A Boolean value that, when TRUE, indicates that the target processor is configured with a FPJ11–AA floating-point accelerator.

ROM_RAM

A Boolean value that, when TRUE, indicates that the target is a ROM-based system.

Example

```
%INCLUDE 'MISC.PAS'

VAR
  Freq : INTEGER;
  Info : HARDWARE_CONFIG;

PROCESS A;
BEGIN

  (* Get the clock frequency and the hardware configuration info *)
  GET_CONFIG (CLOCK_FREQ := Freq,
              CONFIG := Info);

END; (* Process A *)
```

Semantics

The GET_CONFIG procedure obtains information about the target from the kernel's hardware configuration words.

This request is implemented through the GVAL$ kernel primitive.

Error Returns

None

20.6 POWER_FAIL

MACRO equivalent: PWFL$

The POWER_FAIL function allows a process to determine when the kernel has resumed operation after a power failure condition. The function returns a Boolean value to signify whether a power failure has occurred. TRUE indicates that the kernel has resumed operation after a power failure (warm start). FALSE indicates application initialization (cold start), implying that all read/write memory has been cleared by the kernel's initialization routine, as is always the case for an initial system startup. POWER_FAIL is intended for use in the initialization code of a static process that implements some form of powerfail recovery through checkpointing techniques.

A warm restart following a power failure differs from a cold start or cold restart only to the extent that any nonvolatile RAM allocated to a process's impure-data segment is not reinitialized by the kernel during the restart. (That implies that some valid user data may be preserved across the power failure and subsequent power-up, although all kernel data structures are lost and all static processes restarted "from scratch.") Warm restarts are possible only under the following conditions:

- Some or all of the target RAM is declared as nonvolatile in the MEMORY configuration macros (volatile=NO) and is implemented with battery backup. (If you are debugging under PASDBG and only simulating power failures for testing purposes, the RAM in question need not be nonvolatile in actuality but must be declared as such.)
- All code and pure data must reside in nonvolatile memory, whether RAM or ROM.
- The kernel's impure-data area must reside in nonvolatile RAM so the restart indicators are preserved across the power failure, although the area is almost entirely reinitialized on any restart.
- The impure-data area of any process owning data involved in powerfail recovery must reside in nonvolatile RAM.

POWER_FAIL will invariably return a FALSE (cold start) indication if none of the target RAM is declared as nonvolatile in the system configuration file, regardless of actual or simulated power failures. Therefore, the use of POWER_FAIL is meaningful in a simulated, debugging situation only if at least condition 1 is satisfied and is meaningful in actual stand-alone operation only if all the stated conditions are satisfied.

Syntax

POWER_FAIL

Example

```
%INCLUDE 'MISC.PAS'

VAR
  First_time : BOOLEAN;

[INITIALIZE] PROCEDURE Init;
BEGIN

  (* Determine if this is an initial power-up. *)
  First_time := NOT POWER_FAIL;

END; (* Procedure Init *)
```

Semantics

The POWER_FAIL function returns to the caller, with the value FALSE if a bootstrap or the kernel has cleared all read/write memory during the latest system startup or restart. Alternatively, POWER_FAIL returns the value TRUE if user-process data segments allocated in nonvolatile RAM have not been cleared during the latest restart. (The kernel's restart indicators are not reset by the operation of POWER_FAIL.)

This request is implemented through the PWFL$ kernel primitive.

Error Returns

None

Application Notes

POWER_FAIL enables applications that use nonvolatile RAM memories to preserve data when a power failure occurs.

A typical use of POWER_FAIL would be to invoke it from an initialization procedure to enable application software to distinguish between a startup and a resumption after a power failure. If it returns a FALSE, the first-time initialization code is executed. If it returns a TRUE, the applicable code can be executed to resume operation.

Application checkpointing can be accomplished by prudent use of data structures in nonvolatile RAM. Variables that you may wish to use for data-recovery indicators must be defined with the STATIC and VOLATILE attributes at the outermost program level so they will be statically allocated.

You may select the following power-up options with configuration file macros as described in the MicroPower/Pascal system user's guide for your host system:

- Cold start (volatile memory)—All memory is reinitialized, and the application is restarted from the kernel initialization code. All dynamic processes and structures are lost. Static processes are recreated at their initialization code.
- Warm start (nonvolatile memory)—The application is restarted from the kernel-initialization code. However, RAM that is not allocated for the kernel pool (or code) but which is declared as nonvolatile by the MEMORY macro in the configuration file will not be initialized during a power-up following a power failure.

  Volatile and nonvolatile memory may be mixed in this configuration, but the kernel pool must reside in nonvolatile memory (kernel indicators must be preserved).

20.7 TRANSLATE_LOGICAL_NAME

MACRO equivalent: TRLN$

The TRANSLATE_LOGICAL_NAME procedure lets the caller obtain the translation string defined for a given logical name. More precisely, TRANSLATE_LOGICAL_NAME returns the translation string contained in the logical name kernel data structure identified in the call to a specified variable.

TRANSLATE_LOGICAL_NAME, unlike most real-time procedures that operate on existing kernel structures, performs only one level of translation in the case of "nested" logical-name definitions; the immediate translation value is always returned.

The complementary CREATE_LOGICAL_NAME procedure defines the translation value associated with a logical name, and the DELETE_LOGICAL_NAME procedure eliminates the translation value associated with a currently defined logical name.

Syntax

```
TRANSLATE_LOGICAL_NAME ( LENGTH := translation-string-length
                         STRING := translation-string
                         { DESC := logical-name-descriptor }
                         { NAME := logical-name            }
                         [[ STATUS := status-record ]] )
```

translation-string-length

A variable of predefined type LOGICAL_NAME_LEN that specifies the maximum length, in bytes, of the character string being returned to the variable specified by the translation-string parameter. That variable is updated to reflect the actual length of the returned translation string. The valid range of the returned value is 1 to 256.

translation-string

The identifier of a variable that is to receieve the ASCII character string defined as the translation value for the logical name. (The effective length of the string being returned is determined by the translation-string-length parameter.)

logical-name-descriptor

The identifier of a variable of predefined type STRUCTURE_DESC that contains the logical name's structure identifier.

logical-name

A character-string constant or the identifier of a variable of predefined type NAME_STR that specifies the 6-character name of the logical name structure (see Section 11.1.1.1).

status-record

The identifier of a variable of predefined record type EXC_STATUS that may receive an exception type and code. If you specify this parameter, the exception status, either success or error, that results from issuing the request is reported in this variable. Otherwise, an error causes the corresponding exception to be reported. The format of the exception record is described in Section 11.1.2.

Caution

Logical names are, by nature, often dynamically redefined. Since each redefinition can cause the creation of a new logical-name structure, a reference to the logical-name structure, using a descriptor (DESC parameter) may access an obsolete structure.

DIGITAL recommends as a safe programming practice that you refer to logical-name structures by name (NAME parameter) rather than by descriptor to avoid an accidental reference to an obsolete logical-name structure.

Example

```
%INCLUDE 'LOGNAM.PAS'

VAR
  Temp_file : STRUCTURE_DESC;
  Len : LOGICAL_NAME_LEN;
  Str : PACKED ARRAY [1..256] OF CHAR;

[INITIALIZE] PROCEDURE Init;
BEGIN

  (* Translate a logical name. *)
  TRANSLATE_LOGICAL_NAME (DESC := Temp_file,
                          STRING := Str,
                          LENGTH := Len);

END; (* Procedure Init *)
```

Semantics

The TRANSLATE_LOGICAL_NAME procedure verifies that the kernel data structure identified in the call is a logical name structure and tests that the specified maximum length value is at least equal to the length of the translation string. If no error is encountered, the procedure copies the translation string to the caller's buffer variable, places the actual string length in the variable specified by the translation-string-length parameter, and returns to the caller.

This request is implemented through the TRLN$ kernel primitive.

Error Returns

See Section 11.2 for general information about error returns. The following exception codes may be returned:

ES$CDN	(type: SYSTEM_SERVICE)—Cannot specify both descriptor and name
ES$IPM	(type: SYSTEM_SERVICE)—Illegal parameter; the specified maximum string length is less than the actual string length
ES$IST	(type: SYSTEM_SERVICE)—Invalid structure description (index or name); no such logical name exists
ES$MDN	(type: SYSTEM_SERVICE)—Must specify descriptor or name

The request may return the following error, though not as a result of standard Pascal programming practice:

ES$IAD	(type: SYSTEM_SERVICE)—Invalid address; pointer to buffer or structure is odd or not in user address space

Appendix A

ASCII Character Set

Table A–1 lists the standard ASCII character set used by the MicroPower/Pascal software.

Table A–1: ASCII Character Set

Code			Code			Code			Code		
Dec	Oct	Chr	Dec	Oct	Chr	Dec	Oct	Chr	Dec	Oct	Chr
000	000	NUL	016	020	DLE	032	040	SP	048	060	0
001	001	SOH	017	021	DC1	033	041	!	049	061	1
002	002	STX	018	022	DC2	034	042	″	050	062	2
003	003	ETX	019	023	DC3	035	043	#	051	063	3
004	004	EOT	020	024	DC4	036	044	$	052	064	4
005	005	ENQ	021	025	NAK	037	045	%	053	065	5
006	006	ACK	022	026	SYN	038	046	&	054	066	6
007	007	BEL	023	027	ETB	039	047	′	055	067	7
008	010	BS	024	030	CAN	040	048	(	056	070	8
009	011	HT	025	031	EM	041	051	)	057	071	9
010	012	LF	026	032	SUB	042	052	*	058	072	:
011	013	VT	027	033	ESC	043	053	+	059	073	;
012	014	FF	028	034	FS	044	054	,	060	074	<
013	015	CR	029	035	GS	045	055	-	061	075	=
014	016	SO	030	036	RS	046	056	.	062	076	>
015	017	SI	031	037	US	047	057	/	063	077	?

Table A–1 (Cont.): ASCII Character Set

Code			Code			Code			Code		
Dec	**Oct**	**Chr**	**Dec**	**Oct**	**Chr**	**Dec**	**Oct**	**Chr**	**Dec**	**Oct**	**Chr**
064	100	@	080	120	P	096	140	`	112	160	p
065	101	A	081	121	Q	097	141	a	113	161	q
066	102	B	082	122	R	098	142	b	114	162	r
067	103	C	083	123	S	099	143	c	115	163	s
068	104	D	084	124	T	100	144	d	116	164	t
069	105	E	085	125	U	101	145	e	117	165	u
070	106	F	086	126	V	102	146	f	118	166	v
071	107	G	087	127	W	103	147	g	119	167	w
072	110	H	088	130	X	104	150	h	120	170	x
073	111	I	089	131	Y	105	151	i	121	171	y
074	112	J	090	132	Z	106	152	j	122	172	z
075	113	K	091	133	[	107	153	k	123	173	{
076	114	L	092	134	\	108	154	l	124	174	|
077	115	M	093	135	]	109	155	m	125	175	}
078	116	N	094	136	^	110	156	n	126	176	~
079	117	O	095	137	_	111	157	o	127	177	DEL

Table A–2 lists the control code abbreviations of the nonprinting characters.

Table A-2: Control Code Abbreviations for Nonprinting Characters

Abbreviation	Meaning
ACK	Acknowledge (CTRL/F)
BEL	Audible signal (CTRL/G)
BS	Backspace (CTRL/H)
CAN	Cancel (CTRL/X)
CR	Carriage return (CTRL/M)
DC1	Device control 1 (CTRL/Q)
DC2	Device control 2 (CTRL/R)
DC3	Device control 3 (CTRL/S)
DC4	Device control 4 (CTRL/T)
DEL	Delete
DLE	Data link escape (CTRL/P)
EM	End of medium (CTRL/Y)
ENQ	Enquiry (CTRL/E)
EOT	End of transmission (CTRL/D)
ESC	Escape (CTRL/[)
ETB	End of transmission block (CTRL/W)
ETX	End of text (CTRL/C)
FF	Form feed (CTRL/L)
FS	File separator (CTRL/\)
GS	Group separator (CTRL/])
HT	Horizontal tab (CTRL/I)
LF	Linefeed (CTRL/J)
NAK	Negative acknowledge (CTRL/U)
NUL	Null
RS	Record separator (CTRL/^)
SI	Shift in (CTRL/O)
SO	Shift out (CTRL/N)
SOH	Start of heading (CTRL/A)
SP	Space

Table A-2 (Cont.): Control Code Abbreviations for Nonprinting Characters

Abbreviation	Meaning
STX	Start of text (CTRL/B)
SUB	Substitute (CTRL/Z)
SYN	Synchronous idle (CTRL/V)
US	Unit separator (CTRL/_)
VT	Vertical tab (CTRL/K)

Appendix B
Syntax Summary

This appendix summarizes the syntax of the MicroPower/Pascal language. The metalanguage used to specify the syntax of the language constructs is based on the Bakus-Naur Form (BNF). The notation has been modified from the original to permit greater convenience of description.

Each element of the language is defined in terms of simpler elements. The element being defined is written to the left of the equal (=) symbol, and its definition is written to the right of that symbol.

The metalanguage uses a metasymbology that differs from the conventions used in the rest of this manual. These metasymbols are not part of the MicroPower/Pascal language; they are the following:

Symbol	Meaning
=	Shall be defined to be
>	Shall have as an alternative definition
|	Alternatively
.	End of definition
[x]	0 or 1 instance of x
{x}	0 or more instances of x
(x|y)	Grouping: either x or y
"xyz"	The terminal symbol xyz
character-string	A nonterminal symbol

The remainder of this appendix summarizes MicroPower/Pascal language syntax. For ease of reference, the elements are presented in alphabetical order.

```
actual-parameter =
        [ formal-name ":=" ]
          ( expression |
            variable-access |
            procedure-identifier |
            function-identifier |
            empty-parameter ).

actual-parameter-list =
        "(" actual-parameter { "," actual-parameter } ")" .

adding-operator =
        "+" | "-" | "OR" .

apostrophe-image =
        "''" .

array-type =
        "ARRAY" "[" index-type { "," index-type } "]" "OF"
          ( type-identifier | new-type ) .

array-variable =
        variable-access .

assignment-statement =
        ( variable-access | function-identifier ) ":=" expression .

attribute =
        attribute-identifier [ "(" ( constant | identifier ) ")" ] .

attribute-identifier =
        identifier .

attribute-sequence =
        "[" attribute { "," attribute } "]" .

base-type =
        ordinal-type .

binary-digit =
        "0" | "1" .

binary-digit-sequence =
        binary-digit { binary-digit } .

binary-integer =
        "%" letter-b  "'" binary-digit-sequence "'" .

block =
        declaration-part statement-part .

Boolean-expression =
        expression .

Boolean-type =
        "BOOLEAN" .

bound-identifier =
        identifier .

buffer-variable =
        file-variable "^" .

case-constant =
        constant .
```

```
case-constant-list =
        case-constant { "," case-constant } .

case-index =
        expression .

case-list-element =
        case-constant-list ":" statement .

case-statement =
        "CASE" case-index "OF"
          case-list-element { ";" case-list-element } [ ";" ]
          [ "OTHERWISE"  statement-sequence [ ";" ] ]
          "END" .

character-constant =
        "'" string-element "'" .

character-string =
        "'" { string-element } "'" [ "(" unsigned-integer
        { "'" unsigned-integer } ")" [ { character-string } ] ] .

char-type =
        "CHAR" .

compilation-unit =
        program | module .

component-type =
        type-denoter .

component-variable =
        indexed-variable | field-designator .

compound-statement =
        "BEGIN" statement-sequence "END" .

conditional-statement =
        if-statement | case-statement .

conformant-array-schema =
        packed-conformant-array-schema |
          unpacked-conformant-array-schema  .

constant =
        [ sign ] ( unsigned-number | constant-identifier ) |
        "NIL" | character-constant |
        character-string | structured-constant .

constant-definition =
        identifier "=" constant .

constant-definition-part =
        [ "CONST" constant-definition ";"
          { constant-definition ";" } ] .

constant-element =
        ordinal-constant [ ".." ordinal-constant ] .

constant-identifier =
        identifier .

control-variable =
        entire-variable .
```

```
decimal-digit =
        "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" .

decimal-digit-sequence =
        decimal-digit { decimal-digit } .

declaration-part =
        { label-declaration-part
          constant-definition-part
          type-definition-part
          variable-declaration-part
          procedure-function-process-declaration-part } .

digit-sequence =
        decimal-digit-sequence .

directive =
        "FORWARD" | "EXTERNAL" | "SEQ11" .

domain-type =
        type-identifier .

empty-parameter =
        .

empty-statement =
        .

entire-variable =
        variable-identifier .

enumerated-constant =
        unsigned-number | constant-identifier | "NIL" .

enumerated-type =
        "(" identifier-list ")" .

expression =
        simple-expression [ relational-operator simple-expression ] .

factor =
        variable-access | unsigned-constant | bound-identifier |
        function-designator | set-constructor | "NOT" factor |
        "(" expression ")" { "::" type-identifier } |
        structured-constant .

field-designator =
        record-variable "." field-specifier |
        field-identifier .

field-identifier =
        identifier .

field-list =
        ( fixed-part [ ";" variant-part ] | variant-part ) [ ";" ] .

field-specifier =
        field-identifier .

file-type =
        "FILE" "OF" component-type .

file-variable =
        variable-access .
```

```
final-value =
        expression .

fixed-part =
        record-section { ";" record-section } .

for-statement =
        "FOR" control-variable ":="
          initial-value ( "TO" | "DOWNTO" ) final-value
          "DO" statement .

formal-name =
        identifier .

formal-parameter-list =
        "(" formal-parameter-section
          { ";" formal-parameter-section } ")" .

formal-parameter-section =
        value-parameter-specification |
        variable-parameter-specification |
        procedural-parameter-specification |
        functional-parameter-specification .

function-block =
        block .

function-declaration =
        function-heading directive |
        function-identification function-block |
        function-heading function-block .

function-designator =
        function-identifier [ actual-parameter-list ] .

function-heading =
        [ attribute-sequence ] "FUNCTION" identifier
          [ formal-parameter-list ] ":" result-type ";" .

function-identification =
        "FUNCTION" function-identifier ";" .

function-identifier =
        identifier .

functional-parameter-specification =
        function-heading .

goto-statement =
        "GOTO" label .

hexadecimal-digit =
        "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
        "A" | "B" | "C" | "D" | "E" | "F" |
        "a" | "b" | "c" | "d" | "e" | "f" .

hexadecimal-digit-sequence =
        hexadecimal-digit { hexadecimal-digit } .

hexadecimal-integer =
        "%" letter-x  "'" hexadecimal-digit-sequence "'" .

identified-variable =
        pointer-variable "^" .
```

```
identifier =
        letter { letter | decimal-digit } .

identifier-list =
        identifier { "," identifier } .

if-statement =
        "IF" Boolean-expression
          "THEN" statement
          [ "ELSE" statement ] .

include-directive =
        "%INCLUDE" "'" file-specification "'" .

index-expression =
        expression .

index-type =
        ordinal-type .

index-type-specification =
        identifier ".." identifier ":" ordinal-type-identifier .

indexed-variable =
        array-variable
          "[" index-expression { "," index-expression } "]" .

initial-value =
        expression .

integer-type =
        "INTEGER" .

label =
        unsigned-integer .

label-declaration-part =
        [ "LABEL" label { "," label } ";" ] .

letter =
        "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I" |
        "J" | "K" | "L" | "M" | "N" | "O" | "P" | "Q" | "R" |
        "S" | "T" | "U" | "V" | "W" | "X" | "Y" | "Z" | "a" |
        "b" | "c" | "d" | "e" | "f" ||"g" | "h" | "i" | "j" |
        "k" | "l" | "m" | "n" | "o" | "p" ||"q" | "r" | "s" |
        "t" | "u" | "v" | "w" | "x" | "y" | "z" | "$" | "_"  .

letter-b =
        "b" | "B" .

letter-o =
        "o" | "O" .

letter-x =
        "x" | "X" .

member-designator =
        expression [ ".." expression ] .

module =
        module-heading declaration-part "END" "." .

module-heading =
        [ attribute-sequence ] "MODULE" identifier
          [ "(" program-parameters ")" ] ";" .
```

```
multiplying-operator =
        "*" | "/" | "DIV" |"MOD" | "AND" .

new-ordinal-type =
        enumerated-type | subrange-type .

new-pointer-type =
        "^" domain-type .

new-structured-type =
        [ "PACKED" ] unpacked-structured-type .

new-type =
        new-ordinal-type | new-structured-type | new-pointer-type .

octal-digit =
        "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" .

octal-digit-sequence =
        octal-digit { octal-digit } .

octal-integer =
        "%" letter-o "'" octal-digit-sequence "'" .

ordinal-constant =
        integer=constant | Boolean=constant |
        character-constant | enumerated-constant .

ordinal-type =
        new-ordinal-type | integer-type | unsigned-type |
        Boolean-type | char-type | ordinal-type-identifier .

ordinal-type-identifier =
        identifier .

packed-conformant-array-schema =
        "PACKED" "ARRAY" "[" index-type-specification "]"
          "OF" char-type .

pointer-variable =
        variable-access .

procedural-parameter-specification =
        procedure-heading .

procedure-block =
        block .

procedure-declaration =
        procedure-heading directive |
        procedure-identification procedure-block |
        procedure-heading procedure-block .

procedure-function-process-declaration-part =
        { ( procedure-declaration |
            function-declaration |
            process-declaration ) ";" } .

procedure-heading =
        [ attribute-sequence ] "PROCEDURE" procedure-identifier
          [ formal-parameter-list ] ";" .

procedure-identification =
        "PROCEDURE" procedure-identifier ";" .
```

```
procedure-identifier =
        identifier .

procedure-statement =
        procedure-identifier [ actual-parameter-list ] .

process-block =
        block .

process-declaration =
        process-heading directive |
        process-identification process-block |
        process-heading process-block .

process-heading =
        [ attribute-sequence ] "PROCESS" process-identifier
          [ formal-parameter-list ] ";" .

process-identification =
        [ attribute-sequence ] "PROCESS" process-identifier ";" .

process-identifier =
        identifier .

process-statement =
        process-identifier [ actual-parameter-list ] .

program =
        program-heading program-block .

program-block =
        block .

program-heading =
        [ attribute-sequence ] "PROGRAM" identifier
          [ "(" program-parameters ")" ] ";" .

program-parameters =
        identifier-list .

radix-integer =
         binary-integer | octal-integer | hexadecimal-integer  .

read-parameter-list =
        "(" [ file-variable "," ]
          variable-access { "," variable-access } ")" .

record-section  =
        identifier-list ":" type-denoter .

record-type =
        "RECORD" [ field-list ] "END" .

record-variable =
        variable-access .

record-variable-list =
        record-variable { "," record-variable } .

relational-operator =
        "=" | "<>" | "<" | ">" | "<=" | ">=" | "IN" .

repeat-statement =
        "REPEAT" statement-sequence "UNTIL" Boolean-expression .

repetition-factor =
        unsigned-integer | constant-identifier
```

```
repetitive-statement =
        repeat-statement | while-statement | for-statement .

result-type =
        type-identifier .

scale-factor =
        signed-integer .

set-constructor =
        "[" [ member-designator { "," member-designator } ] "]" .

set-type =
        "SET" "OF" base-type .

sign =
        "+" | "-" .

signed-integer =
        [ sign ] unsigned-integer .

simple-constant =
        [ sign ] ( unsigned-number | constant-identifier ) |
        "NIL" | character-constant |
        character-string .

simple-expression =
        [ sign ] term { adding-operator term } .

simple-statement =
        empty-statement | assignment-statement |
        procedure-statement | process-statement |
        goto-statement .

special-symbol =
        "+" | "-" | "*" | "/" | "=" | "<" | ">" | "[" | "]" | "." |
        "," | ":" | ";" | "^" | "(" | ")" | "<>" | "<=" | ">=" |
        ":" | "=" | ".." | "::" | word-symbol .

statement =
        [ label ":" ] ( simple-statement | structured-statement ) .

statement-part =
        compound-statement .

statement-sequence =
        statement { ";" statement } .

string-character =
        "!" | """ | "#" | "$" | "%" | "&" | "'" | "(" | ")" | "*" |
        "+" | "," | "-" | "." | "/" | "0" | "1" | "2" | "3" | "4" |
        "5" | "6" | "7" | "8" | "9" | ":" | ";" | "<" | "=" | ">" |
        "?" | "@" | "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" |
        "I" | "J" | "K" | "L" | "M" | "N" | "O" | "P" | "Q" | "R" |
        "S" | "T" | "U" | "V" | "W" | "X" | "Y" | "Z" | "[" | "\" |
        "]" | "^" | "_" | "a" | "b" | "c" | "d" | "e" | "f" | "g" |
        "h" | "i" | "j" | "k" | "l" | "m" | "n" | "o" | "p" | "q" |
        "r" | "s" | "t" | "u" | "v" | "w" | "x" | "y" | "z" | "{" |
        "|" | "}" | "~" | space | tab .

string-element =
        apostrophe-image | string-character .

structured-constant =
        type-identifier "(" value {"," value} ")".
```

```
structured-statement =
        compound-statement | conditional-statement |
        repetitive-statement | with-statement .

subrange-type =
        ordinal-constant ".." ordinal-constant .

tag-field =
        identifier .

tag-type =
        ordinal-type .

term =
        factor { multiplying-operator factor } .

type-definition =
        identifier "=" type-denoter ";" .

type-definition-part =
        [ "TYPE" type-definition { type-definition } ] .

type-denoter =
        [ attribute-sequence ] ( type-identifier | new-type ) .

type-identifier =
        identifier .

unpacked-conformant-array-schema =
        "ARRAY"  "[" index-type-specification
          { ";" index-type-specification } "]" "OF"
          ( type-identifier | conformant-array-schema ) .

unpacked-structured-type =
        array-type | record-type | set-type | file-type .

unsigned-constant =
        unsigned-number | character-string | constant-identifier
        character-constant | "NIL" .

unsigned-integer =
        digit-sequence | radix-integer .

unsigned-number =
        unsigned-integer | unsigned-real .

unsigned-real =
        unsigned-integer "." digit-sequence [ "E" scale-factor ] |
        unsigned-integer "E" scale-factor .

unsigned-type =
        "UNSIGNED" .

value =
        "(" value ")" | simple-constant |
        unsigned-integer "OF" value .

value-parameter-default =
        ":=" ( variable-identifier | constant ) .

value-parameter-specification =
        identifier-list ":" [ attribute-sequence ]
        type-identifier [ value-parameter-default ] .
```

```
variable-access =
        ( entire-variable |
          component-variable |
          identified-variable |
          buffer-variable) { variable-selection } .

variable-declaration =
        identifier-list ":" type-denoter ";" .

variable-declaration-part =
        [ "VAR" variable-declaration { variable-declaration } ] .

variable-identifier =
        identifier .

variable-parameter-specification =
        "VAR" identifier-list ":" [ attribute-sequence ]
        ( type-identifier [ variable-parameter-default ] |
        conformant-array-schema ) .

variable-parameter-default =
        ":=" ( variable-identifier | unsigned-integer | "NIL" ) .

variable-selection =
        "::" type-identifier | "^" | "." field-specifier |
        "[" index-expression { "," index-expression } "]" .

variant =
        case-constant-list ":" "(" [ field-list ] ")" .

variant-part =
        "CASE" variant-selector "OF"
          variant { ";" variant } .

variant-selector =
        [ tag-field ":" ] tag-type .

while-statement =
        "WHILE" Boolean-expression "DO" statement .

with-statement =
        "WITH" record-variable-list "DO" statement .

word-symbol =
        "AND" | "ARRAY" | "BEGIN" | "CASE" | "CONST" | "DIV" | "DO" |
        "DOWNTO" | "ELSE" | "END" | "EXTERNAL" | "FILE" | "FOR" |
        "FORWARD" | "FUNCTION" | "GOTO" | "IF" | "IN" | "LABEL" |
        "MOD" | "MODULE" | "NIL" | "NOT" | "OF" | "OR" |
        "OTHERWISE" | "PACKED" | "PROCEDURE" | "PROCESS" | "PROGRAM" |
        "RECORD" | "REPEAT" | "SEQ11" | "SET" | "THEN" | "TO" |
        "TYPE" | "UNTIL" | "VAR" | "WHILE" | "WITH" .

write-parameter =
        expression [ ":" expression [ ":" expression ] ] .

write-parameter-list =
        "(" [ file-variable "," ]
          write-parameter { "," write-parameter } ")" .
```

Appendix C
Compile-Time Options

This appendix describes the compiler options you can include in your Pascal program. These options let you select the various compile-time features of the MicroPower/Pascal compiler. The options are analogous to the command string options described in the appropriate MicroPower/Pascal system user's guide.

You specify these options in the comments section of a compilation unit by placing a dollar sign ($) as the first character of the comment. The dollar sign is followed by an option sequence terminated by a blank (space, tab, or end of line) or by the comment terminator. See Chapter 1 for a description of comments. You must separate each option in a sequence with a comma.

Syntax

$$\left\{ \begin{array}{c} \text{(*} \\ \{ \end{array} \right\} \ \$\ \text{option}\ ,\ldots \ \left\{ \begin{array}{c} \text{*)} \\ \} \end{array} \right\}$$

option
One of the directives in the following list.

Option Name	Result When Selected
INDEXCHECK	Perform array bounds checking at run time.
NOINDEXCHECK	Do not perform array bounds checking (default).
STACKCHECK	Perform procedure stack limit checking at run time.
NOSTACKCHECK	Do not perform procedure stack limit checking (default).
POINTERCHECK	Check for NIL pointers.
NOPOINTERCHECK	Do not check for NIL pointers (default).
LIST	Produce a compilation listing (default). This option is effective only if a listing file has been specified in the command string.

NOLIST	Inhibit printing of a compilation listing.
RANGECHECK	Verify that value assigned to a variable is within range of that variable's type declaration. This option does not check against intermediate overflow in expressions.
NORANGECHECK	Do not issue warnings for values that exceed limits specified in type declarations (default).
STANDARD	Issue warnings for use of features not part of Pascal.
NOSTANDARD	Do not issue warnings for use of features not part of Pascal (default).
MATHCHECK	Check for division by 0.
NOMATHCHECK	Do not check for division by 0 (default).

Appendix D
Predefined Data Types in PREDFL.PAS

This appendix shows declarations for the special data types used with the predeclared procedures and functions (real-time requests) declared in the system %INCLUDE file PREDFL.PAS. The MicroPower/Pascal compiler automatically includes this file for you. Other system %INCLUDE files that the compiler does not automatically include are listed in Appendix I. You may examine those files to determine the data types defined within them.

```
CONST
  QE_LEN = 34;  { Size, in bytes, of value section in a packet. }
  BLANK = ''(0,0,0,0,0,0);

TYPE
  RELATIONTYPE = (DEPENDENT, INDEPENDENT);

  UNIVERSAL = [UNSAFE] INTEGER;

  BYTE_RANGE = 0..255;

  PRIORITY_RANGE = 0..255;

  VAL_DATA_LEN = 0..QE_LEN;

  REF_DATA_LEN = 0..8128; { 8192 - 64 bytes }

  RING_BUFFER_DATA = 0..8128;

  RING_BUFFER_SIZE = 8..8128;

  NAME_STR = PACKED ARRAY [1..6] OF CHAR;

  BIN_SEM_VAL = 0..1;

  QUAD_WORD = RECORD
                ONE   : UNSIGNED;
                TWO   : UNSIGNED;
                THREE : UNSIGNED;
                FOUR  : INTEGER;
              END;

  PHYSICAL_ADDRESS = PACKED RECORD
                       ADDRESS   : UNSIGNED;
                       PAR_VALUE : UNSIGNED; { 22-bit addressing }
                     END;
```

```
STRUCTURE_ID = RECORD
                 INDEX         : UNSIGNED;
                 SERIAL_NUMBER : LONG_INTEGER;
               END;

STRUCTURE_DESC = RECORD
                   ID   : STRUCTURE_ID;
                   NAME : NAME_STR;
                 END;

STRUCTURE_DESC_PTR = ^STRUCTURE_DESC;

PROCESS_DESC = STRUCTURE_DESC;

SEMAPHORE_DESC = STRUCTURE_DESC;

QUEUE_SEMAPHORE_DESC = STRUCTURE_DESC;

RING_BUFFER_DESC = STRUCTURE_DESC;

PROCESS_STATE = ( RUN, READY_ACTIVE, READY_SUSPENDED,
                  WAIT_ACTIVE, WAIT_SUSPENDED,
                  EXCEPTION_WAIT_ACTIVE, EXCEPTION_WAIT_SUSPENDED );

PROCESS_TYPE = ( GENERAL, PRIVILEGED, DEV_ACCESS, DRIVER );

EXC_GROUP = 0..255;

EXCEPTIONS = ( MEMORY_FAULT, ILLEGAL_OPERATION, EMULATOR_TRAP,
               TRAP, BREAKPOINT_TRAP, HARD_IO, SOFT_IO, NUMERIC,
               RESOURCE, RANGE, EXECUTION, SYSTEM_SERVICE,
               RESERVED_1, RESERVED_2, USER_1, USER_2 );

EXC_SET = PACKED SET OF EXCEPTIONS;

EXC_STATUS = RECORD
               EXC_TYPE : EXC_SET;
               EXC_CODE : UNSIGNED;
             END;

STATE_CODE_MODIFIER_TYPE = PACKED RECORD
                             RES1,RES2,      { reserved }
                             FPA_PENDING,
                             BLOCKED_ON_COMPLEX,
                             ABORT_TO_INACTIVE,
                             UNBLOCK_IN_PROGRESS,
                             ABORT_PENDING,
                             ABORTED : BOOLEAN;
                           END;

STATE_BLOCK = PACKED RECORD
                PRIORITY            : [BYTE] PRIORITY_RANGE;
                STATE               : [BYTE] PROCESS_STATE;
                TYP                 : [BYTE] PROCESS_TYPE;
                STATE_CODE_MODIFIER : [BYTE] STATE_CODE_MODIFIER_TYPE;
                GROUP               : [BYTE] EXC_GROUP;
                RESERVED            : [BYTE] BYTE_RANGE;
                BLOCKING_SEMAPHORE  : UNIVERSAL;
                SUSPEND_COUNT       : INTEGER;
              END;

QUEUE_MESSAGE = PACKED ARRAY [1..QE_LEN] OF BYTE_RANGE;
```

```
PACKET = PACKED RECORD
           LINK      : UNSIGNED;
           AUXILIARY : UNSIGNED;
           PRIORITY  : [BYTE] PRIORITY_RANGE;
           CONTROL   : [BYTE] BYTE_RANGE;
           MESSAGE   : QUEUE_MESSAGE;
         END;

QUEUE_PTR = ^PACKET;

INFO_BLOCK = PACKED RECORD
               PRIORITY     : [BYTE] PRIORITY_RANGE;
               VAL_XMIT_LEN : [BYTE] VAL_DATA_LEN;
               ADDRESS      : PHYSICAL_ADDRESS;
               REF_XMIT_LEN : [WORD] REF_DATA_LEN;
             END;

ORDERING = ( FIFO, PRIO );

MODE = ( STREAM_MODE, RECORD_MODE );
```

Appendix E

Storage Allocation Rules for Standard Data Types

This appendix describes the storage allocation rules for each of the Pascal data types. All references to storage are in the context of the PDP–11 memory structure: a word occupies 16 bits of memory, and a byte occupies 8 bits. Variables described as occupying a word always start on a word (even address) boundary. Variables described as occupying a byte start on a byte (odd or even address) boundary. Variables described as occupying a bit or a sequence of bits always start on the next bit, as appropriate for the type.

For many of the types, the amount of storage allocated will vary, depending on whether the PACKED modifier was specified in the type definition.

E.1 Scalar Types

The scalar types are the INTEGER, LONG_INTEGER, UNSIGNED, REAL, CHAR, and BOOLEAN, as well as their enumerated and subrange variations. A variable of a scalar type that is not a component of a structured type can be stored only in an unpacked form. This optimizes access time rather than storage allocation. A scalar variable that is a component of a structured type can be stored in either an unpacked or a packed form.

The following sections describe the storage allocation for both unpacked and packed scalar variables.

E.1.1 INTEGER and UNSIGNED Types (unpacked)

A variable of type INTEGER or type UNSIGNED occupies one word. If the variable is a subrange of these types, it is also stored in one word. If the lower bound of a subrange is 0 or positive, the compiler treats the variable as an UNSIGNED type; in this case, the upper bound of the range may be declared as high as 65535. If the lower bound of a subrange is negative, the variable is treated as an INTEGER type; in this case, the bounds are from –32768 to 32767.

E.1.2 INTEGER and UNSIGNED Types (packed)

A variable of type INTEGER or type UNSIGNED that resides in a structure declared to be packed occupies a word. However, if the variable is a subrange of these types, it occupies only the number of bits necessary to represent the minimum and maximum values of the subrange. The subrange is positioned on any bit, as long as the component does not overlap a word boundary. Table E–1 shows the storage allocation for each subrange type.

Table E–1: Storage Allocation for Packed Integer and Unsigned Subrange Types

Integer Subrange	Unsigned Subrange	Storage (in bits)
–1..0	0..1	1
–2..1	0..3	2
–4..3	0..7	3
–8..7	0..15	4
–16..15	0..31	5
–32..31	0..63	6
–64..63	0..127	7
–128..127	0..255	8
–256..255	0..511	9
–512..511	0..1023	10
–1024..1023	0..2047	11
–2048..2047	0..409	12
–4096..4095	0..8191	13
–8192..8191	0..16383	14
–16384..16383	0..32767	15
–32768..32767	0..65535	16

E.1.3 LONG_INTEGER Type (unpacked and packed)

A variable of type LONG_INTEGER always occupies two words, regardless of whether it is packed or unpacked. Subranges of this type are not allowed.

E.1.4 BOOLEAN Type (unpacked)

A variable of type BOOLEAN occupies the least significant bit of a byte. A 0 indicates false, and a 1 indicates true. The remaining seven bits of the byte yield undefined results.

E.1.5 BOOLEAN Type (packed)

A variable of type BOOLEAN in a packed structure is stored in a single bit. A 0 indicates false, and a 1 indicates true.

E.1.6 REAL Type (packed or unpacked)

A variable of type REAL occupies two words, whether or not it resides in a packed structure. The format is that of a standard PDP–11 floating-point number, as described in the *PDP–11 Microcomputer Processor Handbook.*

E.1.7 Enumerated Type (unpacked)

A variable of an enumerated type occupies a byte if the variable has no more than 256 items; otherwise, it occupies a word.

E.1.8 Enumerated Type (packed)

A variable of an enumerated type or a subrange of an enumerated type in a packed structure occupies the minimum number of bits required to represent the number of items in the enumeration. This is equivalent to the storage requirements for an unsigned integer subrange as shown in Table E–1.

E.1.9 CHAR Type (unpacked)

A variable of type CHAR or a subrange of base type CHAR occupies one byte.

E.1.10 CHAR Type (packed)

A variable of type CHAR or a subrange of base type CHAR in a packed structure is stored according to the rules for packed integers (see Section E.1.2). Most of the printing characters have large enough ordinal (ASCII) values so little is gained by packing them. Although nothing is gained in the case of arrays, there is some advantage in the case of small ASCII values; for example, the subrange "(0).."(3) will be packed into two bits, just as would the integer subrange 0..3.

E.2 Pointer Types

Pointer types are stored in a word and are unaffected by packing.

E.3 Structured Types

The structured types (ARRAY, RECORD, and SET) are composed of elements of scalar, pointer, or structured base types according to the rules presented in Chapter 2. Structured types can be stored either in a packed or an unpacked form as controlled by the PACKED modifier in the type definition for the structure. Section E.1 describes the allocation for the scalar components of packed and unpacked structured types.

Note

Defining a structure to be packed may significantly increase the size of the object code generated to access the structure's data elements.

E.3.1 Array Type

An array is stored in successive memory locations according to the row major order sequence of its dimensions. Thus, in a single-dimensional array, the lowest (ordinal) array index value addresses the first element in the list, and the highest index value addresses the last element in the list. For example:

```
VAR  matrix : ARRAY [1..10] OF INTEGER
```

In this example, the array's elements will be stored in 10 successive locations in memory. Because the element type is integer, the location size is one word.

```
matrix[1]     (1st word)
matrix[2]     (2nd word)
matrix[3]     (3rd word)
   .
   .
   .
matrix[10]    (10th word)
```

For each index of a multidimensional array, the lowest (ordinal) index value addresses the first element in the list, and the highest index value addresses the last element in the list.

A multidimensional array is considered to be a series of single-dimensional arrays, each element of which is an array of the succeeding element type. Thus, in a 2-dimensional array, the second array index definition is the component type of the first array; each element of the list of the first array index would be an array of the second element, and so forth. For example:

```
VAR  matrix : ARRAY [1..10] OF ARRAY [1..2] OF INTEGER ;
```

In this example, the following array elements will be allocated 20 words of memory:

```
matrix[1, 1]    (1st word)
matrix[1, 2]    (2nd word)
matrix[2, 1]    (3rd word)
matrix[2, 2]    (4th word)
matrix[3, 1]    (5th word)
    .
    .
    .
matrix[10, 2]   (20th word)
```

The amount of storage required for an array depends on the number of elements, the element type, and whether the array is packed. When an array is not packed, the storage required is the sum of the number of elements, multiplied by the storage allocation for the element type. When an array is packed, the storage size, in bits, of an element is established as follows:

- If the element's base type storage size is 16 bits or less, the packed storage size for the element is the smallest power of 2 that can contain the element.
- If the element's base type storage size is 16 bits or more, the packed storage size for the element is the next larger multiple of 16.

To calculate the amount of storage for an array, determine the number of elements in the array as follows:

1. Multiply those values to obtain the total number of elements in the array.

2. Multiply the result by the storage size of a single element.

This calculation is illustrated by the following formula:

```
(U1 - L1 + 1) * (U2 - L2 + 1) ... * elsiz
```

U
: The upper subscript bound.

L
: The lower subscript bound.

elsiz
: The storage allocation, in bits, bytes, or words, as applicable, for a member of the element type.

Examples

1. Although an element of the following array requires only 5 bits, its size is rounded to 8, and the elements are allocated on byte boundaries:

   ```
   x :  PACKED ARRAY [1..10] OF 0..31;
   ```

2. The following two arrays are equivalent:

   ```
   y :  PACKED ARRAY [1..10,1..3] OF CHAR;
   ```

   ```
   y :  PACKED ARRAY [1..10] OF PACKED ARRAY [1..3] OF CHAR;
   ```

   Each character requires 8 bits, but the element PACKED ARRAY [1..3] OF CHAR requires 24 bits, which is rounded to 32 bits. Thus, each element of the first array ([1..10]) requires 32 bits.

3. An array of elements of type CHAR occupies successive bytes of memory, regardless of whether the array is PACKED. However, the array must be declared PACKED and have a lower bound of 1 if the array is to be type compatible with a string constant of the same length.

   ```
   VAR string :  PACKED ARRAY [1..10] OF CHAR;
   ```

4. An array of elements of type BOOLEAN occupies successive bytes of memory if the array is not PACKED.

   ```
   VAR switches :  ARRAY [1..10] OF BOOLEAN;
   ```

5. In a PACKED array of elements of type BOOLEAN, the bits are mapped into successive bits of a word, from right to left—that is, from low order to high order. This example specifies the bounds of the PACKED array of Boolean elements so that the array index corresponds to the conventional numbering of the bits in the word. But as shown in the next example, this is not required.

   ```
   VAR bit_string :  PACKED ARRAY [0..15] OF BOOLEAN;
   ```

6. In the record one_word, field A occupies the low-order byte of a word; field B, a 2-dimensional array of Boolean elements (bits), occupies the left, or high-order, byte of the word.

```
VAR one_word :  PACKED RECORD
           A :  CHAR;
           B :  PACKED ARRAY [1..4,1..2] OF BOOLEAN;
```

The mapping of array elements onto the bits of the word is as follows:

Bit Number	Array Element
8	b[1,1]
9	b[1,2]
10	b[2,1]
11	b[2,2]
.	.
.	.
.	.
15	b[4,2]

E.3.2 RECORD Type

In general, a record occupies an amount of storage equal to the sum of the amount of storage required for each of its component fields. For example:

```
MIXED = PACKED RECORD
          BITS : PACKED ARRAY [1..4,1..5] OF BOOLEAN (* 4 bytes *)
          NUM  : REAL (* 4 bytes *)
          CHR  : CHAR (* 1 byte *)
```

This example shows a record declaration that contains fields of different data types. The field BITS is a 2-dimensional Boolean array containing another array (that is, a PACKED ARRAY [1..4] OF PACKED ARRAY OF [1..5] OF BOOLEAN). Since the array is packed, each element of the array described by the second dimension (1..5) requires one bit of storage or a total of five bits for each occurrence of this array. Thus, the storage required for an element of the array described by the first dimension (1..4) is also five bits. Since elements must be allocated on byte boundaries, that value must be rounded up to eight bits (one byte). The total amount of storage for the array BITS is therefore four bytes.

The field NUM occupies two words (four bytes) because numbers of type REAL always require this amount of storage.

The field CHR occupies one byte because it is of type CHAR.

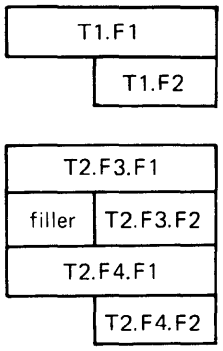
Therefore, an occurrence of the record MIXED will occupy nine bytes of storage.

When using the PACKED modifier on records whose components are records, be aware that a filler byte may be added so a subsequent component begins on a word boundary, as in the following:

Type Definitions **Storage Allocation**

```
TYPE
  T1 = PACKED RECORD
          F1 : INTEGER;
          F2 : CHAR;
       END;

  T2 = PACKED RECORD
          F3, F4 : T1;
       END;
```



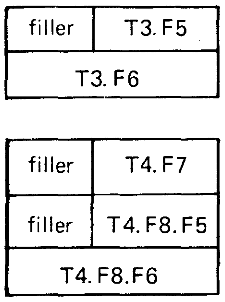
MLO-556-87

Also note that the storage associated with a type is independent of the context in which it is used. See the following example:

Type Definitions **Storage Allocation**

```
TYPE
  T3 = PACKED RECORD
          F5 : CHAR;
          F6 : INTEGER
       END;

  T4 = PACKED RECORD
          F7 : CHAR;
          F8 : T3;
       END;
```



MLO-556A-87

Note

In an application in which both Pascal and MACRO-11 routines share the same data structures, the fields of a packed record that are arrays will begin on a word boundary.

E.3.3 SET Type

Sets occupy storage ranging from 1 byte to 16 words. The amount of storage allocated depends on the ordinal position of the highest value of the base type. Integers, however, are an exception and are allocated 16 words of storage, regardless of the number of elements in the set. The low value has no effect on the storage allocation.' For example, SET OF 0..255 occupies as much space as SET OF 250..255.

The allowable base types and the space occupied by the set are as follows:

Base Type	Maximum Space Required
BOOLEAN	1 byte
Enumerated types	(See Note)
CHAR	16 words
INTEGER (and integer subranges)	16 words
UNSIGNED	16 words

Note

The storage required for an enumerated type is determined by the following formula:

```
storage (bytes) = (ORD(last identifier)+8) DIV 8
```

Appendix F

Summary of Attribute Use

Table F–1 lists the MicroPower/Pascal attributes and shows the language entities with which they may be used. See Chapter 10 for descriptions of the attributes.

Table F–1: Summary of Attribute Use

Attribute	Variable	Formal Param-eter	Record Field	Type Declar.	Routine Declar.	Program/ Module Declar.	Process Declar.
AT (constant)	Yes	No	No	No	No	No	No
BIT ⟦ (constant)⟧	No	No	Yes	No	No	No	No
BYTE ⟦ (constant)⟧	No	No	Yes	No	No	No	No
CONTEXT ($\left\{ \begin{matrix} \text{MMU} \\ \text{FPP} \end{matrix} \right\}$)	No	No	No	No	No	Yes[1]	Yes
DATA_SPACE (constant)	No	No	No	No	No	Yes[1]	No
DEV_ACCESS	No	No	No	No	No	Yes[1]	No
DRIVER	No	No	No	No	No	Yes[1]	No
EXTERNAL ⟦ (global-id)⟧	Yes	No	No	No	Yes	No	Yes
GLOBAL ⟦ (global-id)⟧	Yes	No	No	No	Yes	No	Yes
GROUP (constant)	No	No	No	No	No	Yes[1]	Yes
IDENT (ident-string)	No	No	No	No	No	Yes	No
INITIALIZE	No	No	No	No	Yes[2]	No	No
INIT_PRIORITY	No	No	No	No	No	Yes[1]	No

[1]Not meaningful on module declarations.

[2]Not applicable to function declarations.

Table F-1 (Cont.): Summary of Attribute Use

Attribute	Variable	Formal Param-eter	Record Field	Type Declar.	Routine Declar.	Program/ Module Declar.	Process Declar.
NAME (process-name)	No	No	No	No	No	No	Yes
NOOPTIMIZE	No	No	No	No	Yes	Yes[1]	Yes
OPTIMIZE	No	No	No	No	Yes	Yes[1]	Yes
OVERLAID	No	No	No	No	No	Yes	No
POS (constant)	No	No	Yes[3]	No	No	No	No
PRIORITY (constant)	No	No	No	No	No	Yes[1]	Yes
PRIVILEGED	No	No	No	No	No	Yes[1]	No
READONLY	Yes	Yes	Yes	Yes	No	No	No
STACK_SIZE (constant)	No	No	No	No	No	Yes[1]	Yes
STATIC	Yes	No	No	Yes	No	No	No
SYSTEM (environment-name)	No	No	No	No	No	Yes[1]	No
TERMINATE	No	No	No	No	Yes[2]	No	No
UNSAFE	Yes	Yes	Yes	Yes	No	No	No
VOLATILE	Yes	Yes	Yes	Yes	No	No	No
WORD ⟦ (constant)⟧	No	No	Yes	No	No	No	No
WRITEONLY	Yes	Yes	Yes	Yes	No	No	No

[1] Not meaningful on module declarations.

[2] Not applicable to function declarations.

[3] Applies to packed record fields only.

Appendix G
Predefined Identifiers

This appendix lists the identifiers that are predefined in the MicroPower/Pascal language as the names of files, functions, procedures, types, and values. The standard Pascal identifiers are shown in **boldface** type to differentiate them from the identifiers that denote the extended features of the MicroPower/Pascal language.

The declarations for a number of these identifiers are external to the MicroPower/Pascal compiler and reside in separate system files (see Notes). You must use the %INCLUDE directive to include these files (except PREDFL.PAS) in your program if you intend to use the features defined therein (see Appendix I).

If you redefine an identifier, it no longer has its usual meaning within the scope of the block in which it is redefined. See Section 6.4.1 for a description of the scope of identifiers.

ABORT[2]

ABORTED

ABORT_PENDING

ABS

ACCESS

ACCESS_SHARED_REGION[7]

ACTION

ADDRESS

ADDRESS_SPACE

ADDRESS_TYPE[7]

ADD_CLOCK_TIME[6]

ADD_TIME

ALK

ALLOCATE_PACKET[1]

ALLOCATE_REGION[7]

ARCTAN

ARRAY

AT

[1]Defined in the system file PREDFL.PAS.

[2]Defined in the system file EXC.PAS.

[6]Defined in the system file TIMER.PAS.

[7]Defined in the system file DRAM.PAS.

AUTOEMPTY
AUXILIARY
BIN
BIN_SEM_VAL[1]
BIT
BITNEXT
BITSIZE
BLK
BLOCKING_SEMAPHORE
BOOLEAN
BREAK
BREAKPOINT_TRAP[1]
BUFFERSIZE
BYTE
BYTE_RANGE[1]
CBP
CHANGE_PRIORITY[1]
CHAR
CHR
CLOCK_FREQ
CLOCK_TIME[6]
CLOCK_TIME_USE[6]
CLOCK_VALUE[6]
CLOSE
COD
COMBINE_DATE[6]
COMBINE_TIME[6]
COMMON[7]
COMPLEX_COUNT[5]
COMPLEX_FUNC_VALUE[5]
COND_ALLOCATE_PACKET[1]
COND_ALLOCATE_REGION[7]
COND_GET_ELEMENT[1]
COND_GET_PACKET[1]
COND_PUT_ELEMENT[1]
COND_PUT_PACKET[1]
COND_RECEIVE[1]
COND_RECEIVE_ACK[1]
COND_SEND[1]
COND_SEND_ACK[1]
COND_SIGNAL[1]
COND_WAIT[1]
CONFIG
CONNECT_EXCEPTION[2]
CONNECT_INTERRUPT[1]
CONNECT_SEMAPHORE[1]
CONTEXT
CONTEXT_BLOCK[4]
CONTEXT_SWITCH_OPTIONS[2]
CONTROL
CONTROLLER
COS
COUNT
CREATE_BINARY_SEMAPHORE[1]

[1] Defined in the system file PREDFL.PAS.
[2] Defined in the system file EXC.PAS.
[4] Defined in the system files PCBU.PAS and PCBM.PAS.
[5] Defined in the system file COMPLX.PAS.
[6] Defined in the system file TIMER.PAS.
[7] Defined in the system file DRAM.PAS.

CREATE_BINARY_SEMAPHORE_P[11]
CREATE_COUNTING_SEMAPHORE[1]
CREATE_COUNTING_SEMAPHORE_P[11]
CREATE_LOGICAL_NAME[8]
CREATE_QUEUE_SEMAPHORE[1]
CREATE_QUEUE_SEMAPHORE_P[11]
CREATE_RING_BUFFER[1]
CREATE_RING_BUFFER_P[11]
CREATE_SHARED_REGION[7]
CXW_TYPE[2]
DATA_SPACE
DEALLOCATE_PACKET[1]
DEALLOCATE_REGION[7]
DEFINE_STOP_FLAG[9]
DELETE
DELETE_FILE[3]
DELETE_LOGICAL_NAME[8]
DELETE_SHARED_REGION[7]
DENSITIES[3]
DEPENDENT[1]
DESC
DESTROY[1]
DEV_ACCESS[1]
DIRECT
DISABLE
DISCONNECT_EXCEPTION[2]
DISCONNECT_INTERRUPT[1]
DISCONNECT_SEMAPHORE[1]
DISMISS[2]
DISPOSE
DISPOSITION
DOUBLE[3]
DRIVER[1]
D_SPACE[7]
EMPTY_BUFFER
EMULATOR_TRAP[1]
ENABLE
EOF
EOLN
ES$ABO[10]
ES$ABT[10]
ES$AOV[10]
ES$ASO[10]
ES$ATN[10]
ES$BIV[10]
ES$BOT[10]
ES$BPT[10]
ES$BRK[10]
ES$BUS[10]
ES$CDN[10]
ES$CON[10]
ES$CSO[10]

[1]Defined in the system file PREDFL.PAS.
[2]Defined in the system file EXC.PAS.
[3]Defined in the system file FSINCL.PAS.
[7]Defined in the system file DRAM.PAS.
[8]Defined in the system file LOGNAM.PAS.
[9]Defined in the system file MISC.PAS.
[10]Defined in the system file ESCODE.PAS.
[11]Defined in the system file CRPROC.PAS.

ES$CTL[10]
ES$DAL[10]
ES$DAS[10]
ES$DCF[10]
ES$DDP[10]
ES$DIO[10]
ES$DNU[10]
ES$DRF[10]
ES$DRV[10]
ES$DVF[10]
ES$EMT[10]
ES$EOF[10]
ES$EPN[10]
ES$EVL[10]
ES$EXC[10]
ES$FAO[10]
ES$FDZ[10]
ES$FIV[10]
ES$FNF[10]
ES$FNO[10]
ES$FNR[10]
ES$FNW[10]
ES$FOP[10]
ES$FOR[10]
ES$FOV[10]
ES$FRM[10]
ES$FRO[10]
ES$FUN[10]
ES$FVC[10]
ES$HIO[10]
ES$IAD[10]
ES$IBN[10]
ES$ICD[10]
ES$IDA[10]
ES$IDR[10]
ES$IDS[10]
ES$IDZ[10]
ES$IFN[10]
ES$IFS[10]
ES$IFW[10]
ES$IIV[10]
ES$ILL[10]
ES$ILV[10]
ES$INM[10]
ES$INS[10]
ES$IOP[10]
ES$IOV[10]
ES$IPM[10]
ES$IPR[10]
ES$IRS[10]
ES$IST[10]
ES$IUP[10]
ES$IVC[10]
ES$IVD[10]
ES$IVL[10]
ES$IVM[10]
ES$IVP[10]
ES$KMX[10]
ES$LDZ[10]
ES$LIC[10]
ES$LNM[10]
ES$LNP[10]

[10]Defined in the system file ESCODE.PAS.

ES$LNR[10]

ES$LOV[10]

ES$LRJ[10]

ES$LUC[10]

ES$LUV[10]

ES$MAX[10]

ES$MDN[10]

ES$MEM[10]

ES$MMU[10]

ES$MPT[10]

ES$NFA[10]

ES$NFR[10]

ES$NFS[10]

ES$NID[10]

ES$NIL[10]

ES$NIP[10]

ES$NLZ[10]

ES$NMB[10]

ES$NMC[10]

ES$NMF[10]

ES$NMK[10]

ES$NMP[10]

ES$NMS[10]

ES$NNS[10]

ES$NOR[10]

ES$NRF[10]

ES$NUM[10]

ES$NXM[10]

ES$NXU[10]

ES$OFL[10]

ES$OVF[10]

ES$OVR[10]

ES$PAL[10]

ES$PAR[10]

ES$PCC[10]

ES$PNA[10]

ES$PRO[10]

ES$PWR[10]

ES$RAN[10]

ES$RDE[10]

ES$REF[10]

ES$RNR[10]

ES$RS1[10]

ES$RS2[10]

ES$RSC[10]

ES$RSZ[10]

ES$SEO[10]

ES$SIO[10]

ES$SIU[10]

ES$SNI[10]

ES$SPD[10]

ES$SRN[10]

ES$STO[10]

ES$STU[10]

ES$SVC[10]

ES$TIM[10]

ES$TNF[10]

ES$TRP[10]

ES$UDV[10]

ES$UDZ[10]

ES$UFN[10]

ES$UIV[10]

[10]Defined in the system file ESCODE.PAS.

ES$UNS[10]

ES$UOV[10]

ES$US1[10]

ES$US2[10]

ES$VEC[10]

ES$VSE[10]

ES$WEF[10]

ES$WLK[10]

ESTABLISH[2]

EXCEPTIONS[1]

EXCEPTION_WAIT_ACTIVE[1]

EXCEPTION_WAIT_SUSPENDED[1]

EXC_ACTION[2]

EXC_CODES[2]

EXC_GROUP[1]

EXC_SET[1]

EXC_STATUS[1]

EXECUTION[1]

EXP

EXTERNAL

FALSE

FIFO[1]

FILESIZE

FIND

FIXED[7]

FORMAT_RX02[3]

FPP[2]

FREE[7]

FRIDAY[6]

GENERAL[1]

GET

GET_CONFIG[9]

GET_ELEMENT[1]

GET_ELEMENT_ANY[5]

GET_MAPPING[7]

GET_PACKET[1]

GET_PACKET_ANY[5]

GET_STATE[1]

GET_SYSTEM_DATE_TIME[6]

GET_TIME[6]

GET_VALUE[1]

GLOBAL

GROUP

HARDWARE_CONFIG[9]

HARD_IO[1]

HEX

HISTORY

IAD_NOT_USED[4]

IAD_USED[4]

IDENT

---

[1] Defined in the system file PREDFL.PAS.

[2] Defined in the system file EXC.PAS.

[3] Defined in the system file FSINCL.PAS.

[4] Defined in the system files PCBU.PAS and PCBM.PAS.

[5] Defined in the system file COMPLX.PAS.

[6] Defined in the system file TIMER.PAS.

[7] Defined in the system file DRAM.PAS.

[9] Defined in the system file MISC.PAS.

[10] Defined in the system file ESCODE.PAS.

ID_MAPPING[7]

ILLEGAL_OPERATION[1]

INDEPENDENT[1]

INFO_BLOCK[1]

INITIALIZE

INIT_DIRECTORY[3]

INIT_PRIORITY

INIT_PROCESS_DESC[1]

INIT_STRUCTURE_DESC[1]

INPUT

INTEGER

INTERACTIVE

I_AND_D[4]

I_SPACE[7]

KXT_LOAD & KXJ_LOAD[9]

LN

LOGICAL_NAME_LEN[8]

LONG_INTEGER

LROUND

LTRUNC

MAPPING[7]

MAP_WINDOW[7]

MAXINT

MEMORY_FAULT[1]

MICROPOWER

MMU[2]

MODE[1]

MONDAY[6]

NAME

NAME_STR[1]

NEW

NEXT

NOFPP[2]

NONE

NOOPTIMIZE

NUMERIC[1]

OCT

ODD

OLD

OPEN

OPTIMIZE

ORD

ORDERING[1]

OUTPUT

OVERLAID

OVERLAPPED

PACK

PACKET[1]

PAGE

PAR_PDR_ARRAY[4]

PASS[2]

PCB[4]

[1]Defined in the system file PREDFL.PAS.

[2]Defined in the system file EXC.PAS.

[3]Defined in the system file FSINCL.PAS.

[4]Defined in the system files PCBU.PAS and PCBM.PAS.

[6]Defined in the system file TIMER.PAS.

[7]Defined in the system file DRAM.PAS.

[8]Defined in the system file LOGNAM.PAS.

[9]Defined in the system file MISC.PAS.

PCB_POINTER[4]
PHYSICAL[7]
PHYSICAL_ADDRESS[1]
POS
POWER_FAIL[9]
PRED
PRIO[1]
PRIORITY
PRIORITY_RANGE[1]
PRIVILEGED[1]
PROCESS_DESC[1]
PROCESS_MEMORY_MAP[4]
PROCESS_STATE[1]
PROCESS_TYPE[1]
PROTECT_FILE[3]
PURGE
PUT
PUT_ELEMENT[1]
PUT_PACKET[1]
QUAD_WORD[1]
QUEUE_MESSAGE[1]
QUEUE_PTR[1]
QUEUE_SEMAPHORE_DESC[1]
RANGE[1]
READ
READLN
READONLY
READWRITE
READY_ACTIVE[1]
READY_SUSPENDED[1]
READ_ONLY[7]
READ_WRITE[7]
REAL
RECEIVE[1]
RECEIVE_ACK[1]
RECEIVE_ANY[5]
RECEIVE_ANY_ACK[5]
RECORD_MODE[1]
REF_DATA_LEN[1]
REGION_ID_BLOCK[7]
REGISTER_RANGE[7]
RELATIONTYPE[1]
RELEASE_EXCEPTION[4]
RENAME_FILE[3]
REPORT[2]
RESERVED_1[1]
RESERVED_2[1]
RESET
RESET_RING_BUFFER[1]
RESOURCE[1]
RESTORE_CONTEXT[7]
RESUME[1]
REVERT[2]
REWRITE

[1]Defined in the system file PREDFL.PAS.
[2]Defined in the system file EXC.PAS.
[3]Defined in the system file FSINCL.PAS.
[4]Defined in the system files PCBU.PAS and PCBM.PAS.
[5]Defined in the system file COMPLX.PAS.
[7]Defined in the system file DRAM.PAS.
[9]Defined in the system file MISC.PAS.

RING_BUFFER_DATA[1]
RING_BUFFER_DESC[1]
RING_BUFFER_SIZE[1]
ROUND
RUN[1]
SATURDAY[6]
SAVE
SAVE_CONTEXT[7]
SCHEDULE[1]
SEMAPHORE_DESC[1]
SEND[1]
SEND_ACK[1]
SEQUENTIAL
SET_SYSTEM_DATE_TIME[6]
SET_TIME[6]
SHORT
SIGNAL[1]
SIGNAL_ALL[1]
SIN
SINGLE[3]
SIZE
SLEEP[6]
SLEEP_INTERVAL[6]
SOFT_IO[1]
SPLIT_DATE[6]
SPLIT_TIME[6]
SQR
SQRT
SQUEEZE_DIRECTORY[3]
STACK_SIZE
STATE_BLOCK[1]
STATE_CODE_MODIFIER_TYPE[1]
STATIC
STATUS
STOP[1]
STREAM_MODE[1]
STRUCTURE_DESC[1]
STRUCTURE_DESC_PTR[1]
STRUCTURE_ID[1]
SUBTRACT_CLOCK_TIME[6]
SUCC
SUNDAY[6]
SUSPEND[1]
SYSTEM
SYSTEM_DATE[6]
SYSTEM_DATE_TIME[6]
SYSTEM_SERVICE[1]
SYSTEM_TIME[6]
TERMINATE
TEXT
THURSDAY[6]
TRANSLATE_LOGICAL_NAME[8]
TRAP[1]
TRUE
TRUNC
TUESDAY[6]

[1]Defined in the system file PREDFL.PAS.
[3]Defined in the system file FSINCL.PAS.
[6]Defined in the system file TIMER.PAS.
[7]Defined in the system file DRAM.PAS.
[8]Defined in the system file LOGNAM.PAS.

UAND	VAL_DATA_LEN[1]
UNIVERSAL[1]	VOLATILE
UNMAP_WINDOW[7]	WAIT[1]
UNOT	WAIT_ACTIVE[1]
UNPACK	WAIT_ANY[5]
UNPROTECT_FILE[3]	WAIT_EXCEPTION[2]
UNSAFE	WAIT_SUSPENDED[1]
UNSIGNED	WD$FIX[7]
UOR	WD$INS[7]
UPDATE	WD$RO[7]
UROUND	WEDNESDAY[6]
USER_1[1]	WEEK_DAY[6]
USER_2[1]	WORD
USHORT	WRITE
UTRUNC	WRITELN
UXOR	WRITEONLY

[1] Defined in the system file PREDFL.PAS.

[2] Defined in the system file EXC.PAS.

[3] Defined in the system file FSINCL.PAS.

[5] Defined in the system file COMPLX.PAS.

[6] Defined in the system file TIMER.PAS.

[7] Defined in the system file DRAM.PAS.

Appendix H
MicroPower/Pascal Compiler Limitations

This appendix describes the limitations of the MicroPower/Pascal compiler running on RT–11 and RSX–11 host systems. This appendix is intended to help you in selecting application design and implementation schemes consistent with the compiler's architecture and limits. The limitations of the compiler running on VMS host systems are rarely encountered and are not addressed in this appendix, except where table sizes are mentioned in Sections H.2 and H.3.

Note

> The compiler resources discussed in this appendix are used independently of whether the source code resides in one file or is obtained from %INCLUDE files.

The MicroPower/Pascal compiler's compilation capacity is predicated on the availability of the host system's memory for storing various data structures. The way the compiler uses those memory-resident structures may, depending on the program's size and your coding style, warrant attention. During a compilation, free memory is divided among three major structures: the heap, the subprogram table, and the unique identifier table.

H.1 Heap

The heap is the area of memory where a Pascal program obtains space for storage of dynamic data. The compiler obtains storage space from the heap when processing VAR, TYPE, and CONST declarations and character strings. The compiler requests heap space not only for explicit definitions that appear in the TYPE declaration section but also for implicit type definitions used in the VAR and CONST sections and other contexts as well. Declaration for a structured type may result in more than one request for heap space. Although the text of character strings is placed in a mass-storage file, each string has a type descriptor associated with it that is allocated from heap space.

As a program or module is processed, type definitions and implicit type definitions use space from the heap for the duration of the Pascal scope in which they appear. Therefore, if your source code has a preponderance of type definitions and string constants in the same scope, a compilation may fail, because of insufficient heap space.

The scope of a type definition is that portion of the program in which the definition can be legally referenced (see Section 6.4.1). Given two different scopes, either one is completely contained within the other, or they are disjoint. Type definitions in disjoint scopes require heap space at different times during compilation. Type definitions in the same scope require heap space at the same time during compilation. Assume, for example, that two programs A and B are similar in the form and number of their type definitions. Further, assume that program A's type definitions occur at the outermost (program) level, whereas program B's type definitions occur at the local (restricted) level. During compilation, program B will consume less heap space than program A if B's type declarations reside in disjoint scopes.

If a program exceeds the capacity of the heap, the compiler will detect this as a fatal error and issue one of two messages:

?PASCAL-F-Out of memory, near line NNNNNN

or

?PASOTS-F-Stack Overflow

Programs that use the modular features of the language and that limit declarations to the modules and procedure scopes in which they are needed are less likely to cause heap-space problems. However, large applications may have many declarations at the program level. For example, when many modules share data declarations by means of one or more generalized %INCLUDE files. Such applications are more likely to experience capacity overflow.

When a program exceeds compiler capacity, try the following remedies in the order shown:

1. Compile the program, using the /F or /FI (filter declarations) command option. Those options may permit a successful compilation by filtering out the standard declarations for real-time requests that are not referenced in the compilation unit. Those declarations are defined in the system file PREDFL.PAS and are automatically included when you do not use the /N or /NOP command option. (Refer to the applicable MicroPower/Pascal system user's guide for detailed information about the /F and /FI command options.)

2. Consider restructuring the program so the type definitions reside within the most limited (disjoint) scope possible. This tactic requires the careful use of common %INCLUDE files, especially at the program level.

3. If the previous suggestion is not feasible or does not solve the capacity problem, consider rewriting the declarations. Different sorts of declarations affect the heap-space allocation differently. You may be able to replace some declarations with equivalent code that makes more efficient use of the heap space available to the compiler. Information is provided in the examples to demonstrate how different constructs use the heap.

4. If a single procedure contains many strings, you should break up the procedure into two procedures. The disjoint scopes of the procedures will reduce the number of character strings requiring heap space at any one time. If code containing strings is localized and appears at the top level of the program, such as when tables of strings are initialized, you should move that code into a separate procedure. If the capacity problem persists, you may optionally place the procedure in a separate module with an abbreviated declaration section and call externally.

In the following examples of Pascal source code, the circumflex (^) symbol under a code line indicates that space is allocated from the heap during compilation for the indicated construct. Comments explain why the space allocation from the heap is made and what possible alternative coding styles might reduce heap use.

```
Source Code                     Comments

VAR
   i : integer;                 (* "integer" is a predefined type, so no heap *)
                                (* space is required. *)

   j : ARRAY [1..2] OF integer; (* Every "ARRAY" causes a request for heap *)
       ^       ^                (* space, as does every ".." (subrange) construct. *)
TYPE
  colors = (red, white, blue);  (* Generally, every user-defined type results in *)
           ^                    (* at least one request for heap space, as the *)
                                (* simple enumerated scalar type "colors". *)

  rect = RECORD                 (* Every "RECORD" symbol results in a request *)
         ^                      (* for heap space. If you define a type *)
     wall, floor,               (* once and use it again, no additional *)
     ceiling : colors;          (* heap space is used. Each case variant label *)
      CASE paint : colors OF    (* requires space from the heap. *)
       red: (glossy:boolean);
       ^
       white,blue: (spray:boolean)
       ^     ^
      END;

  ptr = ^integer;               (* Every "^" (pointer), SET, and FILE *)
        ^                       (* requires space from the heap. Heap *)
  alpha = SET OF char;          (* storage is required for "0..255".  If *)
            ^                   (* we use this range in other declarations, *)
  file1t = FILE OF 0..255;      (* it is best to give it a type identifier *)
             ^      ^           (* as in this case "byte_rng". *)
  byte_rng = 0..255;
              ^

  file2t = FILE OF byte_rng;    (* Using a type identifier for a subrange *)
           ^                    (* reduces heap use in subsequent occurrences *)
  byteptr = ^byte_rng;          (* of that subrange. *)
            ^
  array_range = 1..10;          (* This applies to any variable or type *)
                 ^              (* component that is used often. *)

  arrayt1 = ARRAY[array_range,array_range,array_range] OF byte_rng;
            ^
VAR
  a1,a2,a3 : arrayt1;           (* Collapsing commonly used structure *)
                                (* components into single type definitions *)
                                (* that are reused reduces heap use. Compare *)
                                (* this simple declaration with the equivalent: *)

  a1 : ARRAY [1..10,1..10,1..10] OF 0..255;
       ^       ^     ^     ^         ^
  a2 : ARRAY [1..10,1..10,1..10] OF 0..255;
       ^       ^     ^     ^         ^
  a3 : ARRAY [1..10,1..10,1..10] OF 0..255;
       ^       ^     ^     ^         ^
```

```
CONST
  one = 1;                      (* As with VAR or TYPE declarations, if *)
                                (* the type of a CONST value is a standard *)
                                (* type (in this case implicitly INTEGER), *)
                                (* no space is allocated from the heap. *)

  ch = 'a';                     (* Standard type "char". *)

  s1 = 'abc';                   (* If a character string appears anyplace -- in *)
       ^^                       (* either a declaration or a statement -- it will *)
                                (* cause two requests for heap space. In this *)
                                (* case, 'abc' is best declared with a constant *)
                                (* identifier if it is used more than once *)
                                (* in statement code. *)
TYPE
   tablet = ARRAY[array_range] OF PACKED ARRAY[1..5] OF char;
            ^                      ^             ^
CONST
   commands = tablet('start','stop ','left ','right','help ',
                      ^^      ^^      ^^      ^^      ^^
                     'revrs','fast ','slow ','signl','beep ');
                      ^^      ^^      ^^      ^^      ^^
                                (* Generous use of strings causes *)
                                (* extensive use of the heap. *)

VAR                             (* The same problem occurs when strings *)
  command : tablet;             (* appear in statement code. *)

  { statement code }
BEGIN
   command := tablet('start','stop ','left ','right','beep ',
                      ^^      ^^      ^^      ^^      ^^
                     'revrs','fast ','slow ','signl','help ');
                      ^^      ^^      ^^      ^^      ^^
   IF command[1] = 'beep ' THEN { ... } ;
                    ^^
END;
```

Examples

The following examples show how a declaration's scope affects heap usage:

1. This example illustrates a programming technique that causes inefficient use of heap space. Note that the type definitions for Names1 through Names5 have been declared at the outermost (program) level, although several of them could have been placed within a more restricted scope.

```
PROGRAM Test3;
  TYPE
    Names1 = (bob, louise, luke, monica);
    Names2 = (mark, alita, brian, jack);
    Names3 = (norm, brad, bill);
    Names4 = (katie, chrissy, anne, adrian);
    Names5 = (james, richard, tom);
  VAR
    AO: Names4;
```

```
PROCEDURE Z1;
  TYPE
    Names6 = (harpo, grocho, zeppo);
  VAR
    A1: Names5;

  PROCEDURE Z2;
    VAR
      A2: Names1;
    BEGIN {Z2}
    END; {Z2}

  PROCEDURE Z3;
    VAR
      A3: Names2;
      B3: Names6;
    BEGIN {Z3}
    END; {Z3}

  PROCEDURE Z4;
    VAR
      A4: Names2;
      B4: Names3;
    BEGIN {Z4}
    END; {Z4}
  BEGIN {Z1}
  END; {Z1}

PROCEDURE Z5;
  VAR
    A5: Names1;
  BEGIN {Z5}
  END; {Z5}

BEGIN {test3}
END. {test3}
```

2. This example illustrates how the type definitions shown in example 1 could be placed to cause efficient use of heap space during compilation. In each case, the type definitions have been relocated to reside in the most restricted scope possible.

```
PROGRAM Test4
  TYPE
    Names1 = (bob, louise, luke, monica);
      {Names2 = (mark, alita, brian, jack);} {moved to procedure Z1}
      {Names3 = (norm, brad, bill);} {moved to procedure Z4}
    Names4 = (katie, chrissy, anne, adrian);
      {Names5 = (james, richard, tom);} {moved to procedure Z1}
  VAR
    A0: Names4;

  PROCEDURE Z1;
    TYPE
      Names2 = (mark, alita, brian, jack); {moved here from outermost level}
      Names5 = (james, richard, tom); {moved here from outermost level}
        {Names 6 = (harpo, grocho, zeppo);} {moved to procedure Z3}
    VAR
      A1: Names5;
  PROCEDURE Z2;
    VAR
      A2: Names1;
    BEGIN {Z2}
    END; {Z2}

  PROCEDURE Z3;
    TYPE
      Names6 = (harpo, groucho, zeppo); {moved here from procedure Z1}
    VAR
      A3: Names2;
      B3: Names6;
    BEGIN {Z3}
    END; {Z3}

  PROCEDURE Z4;
    TYPE
      Names3 = (norm, brad, bill); {moved here from outermost level}
    VAR
      A4: Names2;
      B4: Names3;
    BEGIN {Z4}
    END; {Z4}

  BEGIN {Z1}
  END; {Z1}

PROCEDURE Z5;
  VAR
    A5: Names1;
  BEGIN {Z5}
  END; {Z5}

BEGIN {test4}
END. {test4}
```

H.2 Subprogram Table

The subprogram table has 110 possible entries (300 for MicroPower/Pascal–VMS), one for each procedure, function, or process that you declare in the compilation unit. Table entries will also be allocated for subprograms that are declared through references to %INCLUDE files (including the system %INCLUDE files listed in Appendix I).

If a program exceeds the capacity of this table, the compiler will issue the message "Too many procedures (only 110 allowed)." You can correct this problem by using one or more of the following suggestions:

- Divide the program into separate compilation units (modules)
- Reduce the number of subprograms by combining some of them
- Edit %INCLUDE files to eliminate declarations for subprograms not required by the compilation unit

H.3 Unique Identifier Table

The unique identifier table is used during lexical analysis and contains no information about program context—that is, whether an identifier is associated with a CONST or VAR declaration or a TYPE definition. The table has 997 possible entries (3000 for MicroPower/Pascal–VMS), one for each identifier in the compilation unit. Of these, approximately one-third are used for the identifiers associated with the predeclared subprograms that reside in the system file PREDFL.PAS; the remaining entries are available for user-specified identifiers.

If a program exceeds the capacity of this table, the compiler will issue the message "Too many identifiers (only 997 allowed)." This problem has two possible solutions. You may either divide the program into separate compilation units (modules) or reduce the number of unique identifiers and instead reuse identifiers that are not in the current scope. Note that Pascal scoping rules permit the reuse of identifiers without altering the meaning of a program (see Section 6.4.1).

Example

This example shows how you might rewrite a program (Test1) to reuse identifiers (Test2).

```
PROGRAM Test1;
  CONST
    max_1 = 10;
  VAR
    i : INTEGER;

  PROCEDURE Z1;
    CONST
      max_2 = 20;
    VAR
      j : INTEGER;
    BEGIN {Z1}
      FOR j := 1 TO max_2 DO
        WRITELN (j, SQRT(j));
    END; {Z1}

  BEGIN {Test1}
    FOR i := 1 TO max_1 DO
      WRITELN (i, i*i);
  END. {Test1}

PROGRAM Test2;
  CONST
    max_val = 10;   {replaced max_1 with max_val}
  VAR
    i: INTEGER;
```

```
PROCEDURE Z1;
  CONST
    max_val = 20; {replaced max_2 with max_val}
  VAR
    {j : INTEGER;     replaced j with i}
    i : INTEGER;
  BEGIN {Z1}
    FOR i := 1 TO max_val DO   {changed max_2 to max_val}
      WRITELN (i, SQRT(i));
  END; {Z1}

BEGIN {Test2}
  FOR i := 1 TO max_val DO     {changed max_1 to max_val}
    WRITELN (i, i*i);
END. {Test2}
```

Appendix I

System %INCLUDE and Module Files and Associated Requests

This appendix lists the I/O and real-time requests that are defined in the MicroPower/Pascal system files. For most of those requests, you must use the %INCLUDE directive to include their formal parameter definitions in your program or module before using a request. For some requests, you must also compile and merge with the module that implements the request. The requests defined in PREDFL.PAS are automatically included for you by the compiler.

Note

If you use the OPEN procedure, you must install the ACP when you build your application (as described in the MicroPower/Pascal system user's guide for your host system).

%INCLUDE Files	**Request**	
COMPLX.PAS	GET_ELEMENT_ANY	RECEIVE_ANY
	GET_PACKET_ANY	RECEIVE_ANY_ACK
CRPROC.PAS	CREATE_BINARY_SEMAPHORE_P	CREATE_COUNTING_SEMAPHORE_P
	CREATE_QUEUE_SEMAPHORE_P	CREATE_RING_BUFFER_P
DRAM.PAS	ACCESS_SHARED_REGION	GET_MAPPING
	ALLOCATE_REGION	MAP_WINDOW
	CREATE_SHARED_REGION	RESTORE_CONTEXT
	DEALLOCATE_REGION	SAVE_CONTEXT
	DELETE_SHARED_REGION	UNMAP_WINDOW

%INCLUDE Files	Request	
EXC.PAS[1]	CONNECT_EXCEPTION	REPORT
	DISCONNECT_EXCEPTION	RELEASE_EXCEPTION
	ESTABLISH	REVERT
FSINCL.PAS	DELETE_FILE	RENAME_FILE
	FORMAT_RX02	SQUEEZE_DIRECTORY[2]
	INIT_DIRECTORY[3]	UNPROTECT_FILE
	PROTECT_FILE	
LOGNAM.PAS	CREATE_LOGICAL_NAME	TRANSLATE_LOGICAL_NAME
	DELETE_LOGICAL_NAME	
MISC.PAS	DEFINE_STOP_FLAG	POWER_FAIL
	GET_CONFIG	SET_STOP_FLAG
MUTEX.PAS	CREATE_MUTEX	LOCK_MUTEX
	DESTROY_MUTEX	POWER_FAIL
	DISABLE_STOP	UNLOCK_MUTEX
	ENABLE_STOP	
PREDFL.PAS	ALLOCATE_PACKET	DISCONNECT_SEMAPHORE
	CHANGE_PRIORITY	GET_ELEMENT
	COND_GET_ELEMENT	GET_PACKET
	COND_GET_PACKET	GET_STATUS
	COND_PUT_ELEMENT	GET_VALUE
	COND_PUT_PACKET	INIT_PROCESS_DESC
	COND_RECEIVE	INIT_STRUCTURE_DESC
	COND_RECEIVE_ACK	PUT_ELEMENT
	COND_SEND	PUT_PACKET
	COND_SEND_ACK	RECEIVE
	COND_SIGNAL	RECEIVE_ACK
	COND_WAIT	RESET_RING_BUFFER

[1] The two files PCBM.PAS and PCBU.PAS are process control block (PCB) declaration files. Each file contains a variation of the PCB declaration. You select PCBM.PAS if your application uses memory mapping and PCBU.PAS if it does not use memory mapping. You must include one of these files and EXC.PAS in the exception handler of your program.

[2] You must build your program with module SQUEEZ.PAS to use this feature.

[3] You must build your program with module INTDIR.PAS to use this feature.

%INCLUDE Files	Request	
	CONNECT_INTERRUPT	RESUME
	CONNECT_SEMAPHORE	SCHEDULE
	CREATE_BINARY_SEMAPHORE	SEND
	CREATE_COUNTING_SEMAPHORE	SEND_ACK
	CREATE_QUEUE_SEMAPHORE	SIGNAL
	CREATE_RING_BUFFER	SIGNAL_ALL
	DEALLOCATE_PACKET	STOP
	DESTROY	SUSPEND
	DISCONNECT_INTERRUPT	WAIT
TIMER.PAS	ADD_TIME	SET_SYSTEM_DATE_TIME
	COMBINE_DATE	SET_TIME
	COMBINE_TIME	SPLIT_DATE
	GET_SYSTEM_DATE_TIME	SPLIT_TIME
	GET_TIME	SUBTRACT_TIME
	SLEEP	

Index

HOW TO ORDER ADDITIONAL DOCUMENTATION

From	Call	Write
Alaska, Hawaii, or New Hampshire	603–884–6660	Digital Equipment Corporation P.O. Box CS2008 Nashua, NH 03061
Rest of U.S.A. and Puerto Rico*	800–258–1710	
* Prepaid orders from Puerto Rico must be placed with DIGITAL's local subsidiary (809–754–7575)		
Canada	800–267–6219 (for software documentation) 613–592–5111 (for hardware documentation)	Digital Equipment of Canada Ltd. 100 Herzberg Road Kanata, Ontario, Canada K2K 2A6 Attn: Direct Order desk
Internal orders (for software documentation)	—	Software Distribution Center (SDC) Digital Equipment Corporation Westminster, MA 01473
Internal orders (for hardware documentation)	617–234–4323	Publishing & Circulation Serv. (P&CS) NR03–1/W3 Digital Equipment Corporation Northboro, MA 01532

READER'S COMMENTS

Note: This form is for document comments only. DIGITAL will use comments submitted on this form at the company's discretion. If you require a written reply and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Did you find this manual understandable, usable, and well organized? Please make suggestions for improvement.

Did you find errors in this manual? If so, specify the error and the page number.

Please indicate the type of user/reader that you most nearly represent:

- ☐ Assembly language programmer
- ☐ Higher-level language programmer
- ☐ Occasional programmer (experienced)
- ☐ User with little programming experience
- ☐ Student programmer
- ☐ Other (please specify) ____________

Name ____________ Date ____________

Organization ____________

Street ____________

City ____________ State ____________ Zip Code or Country ____________

— Do Not Tear — Fold Here and Tape — — — — — — — — — — — — — — — — — —

digital

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO.33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

DIGITAL EQUIPMENT CORPORATION
CORPORATE USER PUBLICATIONS
MLO5–5/E45
146 MAIN STREET
MAYNARD, MA 01754–2571

— Do Not Tear — Fold Here —

Cut Along Dotted Line