First Edition - March 1985

This manual is a reference guide to the VAXELN Pascal programming language.

VAXELN Pascal Language Reference Manual

Document Order Number: **AA-EU39A-TE**

Software Version: 2.0

digital equipment corporation
maynard, massachusetts

First Edition, March 1985

The postage-paid READER'S COMMENTS form on the last page of this document requests your critical evaluation to assist us in preparing future documentation.

Contents

Chapter 3: Data Types

Chapter 5: Variables

Chapter 6: Expressions and Operators

Chapter 7: Pascal Statements

Chapter 9: VAXELN Routines

Chapter 14: Device Drivers and Interrupts

Chapter 15: Input and Output

List of Figures

List of Tables

Preface

VAXELN Pascal is a compatible superset of the language defined in the International Standards Organization document ISO DIS 7185. Any program written in ISO-standard Pascal can be compiled by the VAXELN Pascal compiler and executed as part of the system.

However, VAXELN Pascal has been extended to include data types and operations that support concurrent programming. It is supported by a highly optimizing compiler that generates position-independent, native-mode code. In addition, it is the primary implementation language of the VAXELN toolkit itself.

The *VAXELN Pascal Language Reference Manual* is a reference guide describing the elements of the extended VAXELN Pascal programming language and a guide to program development using VAXELN Pascal.

Manual Objectives

This manual is a summary of the VAXELN Pascal language, for daily reference and for review by people already familiar with Pascal. It is not intended to be a tutorial document; however, it presents the features of VAXELN Pascal in detail, and explains how to develop, compile, and link VAXELN Pascal programs for inclusion in a VAXELN system.

Intended Audience

This manual is designed for programmers and students who have a working knowledge of Pascal. Knowledge of the fundamental principles of the VAX/VMS operating system is required, as well as knowledge of VAXELN.

Structure of this Document

This manual consists of 16 chapters and 3 appendices, organized as follows:

- Chapter 1, "Notation and Lexical Elements," explains the source text conventions, syntax conventions, and call format conventions used in VAXELN Pascal.
- Chapter 2, "Program Structure," describes the structure of VAXELN Pascal programs in relation to compilation units, modules, PROGRAM blocks, routine bodies, and the scope of declarations.
- Chapter 3, "Data Types," discusses the declaration of data type names, the definition of each of the VAXELN Pascal data types, type equivalence, and the rules for data representation.
- Chapter 4, "Constants," discusses the rules for literal constants, the declaration of named constants, and initializers.
- Chapter 5, "Variables," discusses the declaration of variables, the rules for variable references, storage allocation, and data sharing between processes in a job.

- Chapter 6, "Expressions and Operators," discusses the syntax of VAXELN Pascal expressions, the operators used in expressions, and the rules for operator precedence and associativity.
- Chapter 7, "Pascal Statements," summarizes the statements available in VAXELN Pascal.
- Chapter 8, "Procedures and Functions," summarizes the rules for declaring and calling procedures and functions in VAXELN Pascal, including parameter/argument relationships and calling conventions.
- Chapter 9, "VAXELN Routines," describes the arithmetic, ordinal, string, argument, and VAX functions available in VAXELN Pascal, as well as the type conversion, storage allocation, time representation, and other VAXELN routines that are available and are not described under specific topics in later chapters.
- Chapter 10, "Queues," discusses queue declarations, queue procedures, and using queues in interprocess communication.
- Chapter 11, "Subprocesses and Synchronization," discusses process blocks and the kernel services relating to processes and synchronization, process UICs and the authorization procedures, the Authorization Service utility, program loader utility, and exit utility procedures, the MUTEX data type, and the VAXELN procedures that perform operations on mutexes.
- Chapter 12, "Interjob Communication," discusses messages and ports and the kernel services relating to message transmission, interjob data sharing and the related kernel services, and the memory allocation and stack utility procedures.

- Chapter 13, "Errors and Exception Handling," discusses errors, the EXCEPTION_HANDLER function type, exception names and status values, and the exception handling procedures.
- Chapter 14, "Device Drivers and Interrupts," discusses device driver programs and the kernel services relating to devices, interrupt service routines, the procedures used to manipulate interrupt priority levels, the procedures relating to direct memory access UNIBUS and QBUS devices, the device register procedures, and the real-time device driver utility procedures.
- Chapter 15, "Input and Output," discusses files and their use in file I/O, record-oriented device I/O, and circuits, Pascal I/O routines, and the VAXELN file utility, disk utility, and tape utility procedures.
- Chapter 16, "Program Development," discusses the format and arguments of the EPASCAL command, as well as module management.
- Appendix A, "Attributes," lists the attributes allowed in VAXELN Pascal and the context in which they may be used.
- Appendix B, "Collected Syntax," is an alphabetical collection of syntax diagrams representing the syntactic categories of the VAXELN Pascal language.
- Appendix C, "Call Formats," lists the call formats of the procedures and functions available in VAXELN Pascal.

Associated Documents

The following documents are relevant to VAXELN Pascal programming:

- *VAXELN Release Notes (AA-Z454C-TE)*
- *VAXELN Installation Manual (AA-EU37A-TE)*
- *VAXELN User's Guide (AA-EU38A-TE)*
- *VAXELN Application Design Guide (AA-EU41A-TE)*
- *VAX/VMS DCL Dictionary (AA-Z200A-TE)*
- *VAX/VMS Run-Time Library Routines Reference Manual (AA-Z502A-TE)*
- *VAX Architecture Handbook (EB-19580-20)*
- *VAX Hardware Handbook 1982-1983 (EB-21812-20)*
- *LSI–11 Analog System User's Guide (EK-AXV11-UG)*

Chapter 1
Notation and Lexical Elements

This manual contains VAXELN Pascal call formats, syntax diagrams, and examples, ranging from simple to complex constructions. Complex examples have been divided into several lines to make them easy to read. Pascal does not require that you format your programs in any particular way; therefore, you should not regard the formats used as mandatory.

This chapter explains the source text conventions, syntax conventions, and call format conventions used in VAXELN Pascal.

Source Text Conventions

The source text of a VAXELN module is an ASCII file, which may include other ASCII files by means of the %INCLUDE construction described later in this section. The compiler performs a two-level structure analysis on the source text. The first level, lexical analysis, divides the text into a sequence of *lexical tokens*: identifiers, reserved words, literal constants, and special symbols. Intervening spaces, comments, and line breaks are ignored after lexical analysis.

The second level of analysis, parsing, determines how the sequence of tokens is structured into language components, such as statements and expressions. Parsing is governed by the syntax diagrams shown throughout this manual and collected in Appendix B. The lexical tokens are the terminal elements of these syntax diagrams; that is, they cannot be broken down any further into other syntax elements. The complete

syntax notation is given in the section "Syntax Conventions," later in this chapter.

The form of the various literal constants is explained in Chapter 4, "Constants." The following subsections describe the other types of lexical tokens and the related rules.

Identifiers

Identifiers are used as the names of variables, constants, types, programs, and so forth. They must conform to the following rules:

- The first character must not be a digit.
- The maximum length of identifiers is 31 characters.
- The set of valid characters consists of the uppercase letters, lowercase letters, digits, underline (_), and dollar sign ($).
- Identically spelled identifiers mean the same thing regardless of the cases of letters; for example, ABC, abc, and aBc all denote the same thing.
- Identifiers must not have the same spellings as the reserved words in Table 1-1, below.

Note that the dollar sign and underline characters may not be allowed in other versions of Pascal. In particular, we recommend that you limit the use of the dollar sign to names of system-specific global values and avoid its use for ordinary program variables.

In syntax diagrams, the word *identifier* means a name conforming to the above rules and is used at the defining point of some entity, such as a data type name. The term *name* indicates a legal identifier that has been declared (or is predeclared) for a particular use.

Reserved Words

The words listed in Table 1-1 are *reserved* and must not be used as identifiers. Note that identically spelled reserved words mean the same thing regardless of the cases of letters; for example, PROGRAM, Program, and program all mean the same thing.

Table 1-1. Reserved Words

and	for	not	set
array	function	of	then
begin	function_body	or	to
case	goto	otherwise	type
const	if	packed	until
div	in	procedure	var
do	interrupt_service	procedure_body	while
downto	label	process_block	with
else	mod	program	
end	module	record	
file	nil	repeat	

Special Symbols

Special symbols represent punctuation marks used as delimiters, as well as arithmetic, relational, and set operators.

Punctuation Symbols

Table 1-2 shows the punctuation symbols used in VAXELN Pascal programs; for some, alternatives are

allowed, which are shown in the table. Note that symbols consisting of two characters must not have embedded spaces, comments, or line breaks.

Note: In this manual, the symbol ↑ is normally used for clarity in pointer type declarations, identified variables, and buffer variables; the character ^ is more commonly available on terminal keyboards. Note also that quotation marks (",") are not valid delimiters of character and string constants; apostrophes (',') must be used.

Operators

Table 1-3 summarizes the special symbols used as arithmetic, relational, or set operators. Both *dyadic* and *monadic* arithmetic operations are defined where appropriate. Monadic operators require a single operand; dyadic operators require two. Note that operators consisting of two characters must not have embedded spaces, comments, or line breaks.

Table 1-3. Special Symbol Operators

Operator	Operation(s)
+	Addition (dyadic); identity (monadic); string concatenation; set union
−	Subtraction (dyadic); sign inversion (monadic); set difference
*	Multiplication (dyadic); set intersection
/	Division (dyadic)
**	Exponentiation
=	Equality
<>	Inequality
<	"Less than"
>	"Greater than"
<=	"Less than or equal to," set inclusion
>=	"Greater than or equal to," set inclusion

Spaces, Comments, and Punctuation Rules

Most of the punctuation rules of VAXELN Pascal are shown by the syntax diagrams, but there are a few rules that apply at the lexical level. Spaces, tabs,

comments, and line breaks may occur before the first lexical token, between two tokens, and after the last token in the source text. None of these may occur within a single token.

A comment is any series of characters enclosed in braces. The comment ends at, and cannot contain, the right brace (}). Comments cannot be nested. Note that the character pairs (* and *) may be used instead of { and }, respectively.

If two adjacent tokens are identifiers, reserved words, literal integer constants, or literal floating-point constants, in any combination, they must be separated by one or more spaces, tabs, comments, or line breaks. Also, two adjacent literal CHAR constants and/or literal string constants must be similarly separated.

The following examples illustrate VAXELN Pascal punctuation rules:

```
PROGRAM F; { Valid. }
PROGRAMF;     { Invalid. }
PROGRAM{ Valid. }F;
PROGRAM
   F; { Valid. }
```

%INCLUDE

The %INCLUDE construction includes a file of Pascal source text in a compilation. It can be used for such purposes as sharing source text with a different Pascal compiler or including source text generated by a "definition language" or other source text generator. The syntax is shown in Figure 1-1 and is valid anywhere that comments, spaces, tabs, or line breaks can occur.

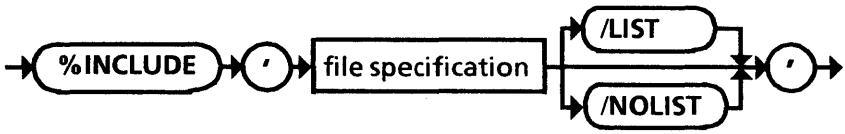


Figure 1-1. %INCLUDE Syntax

The file specification designates a file of Pascal source text; the text is included in the compilation at the location of the %INCLUDE construction. The /LIST and /NOLIST options specify whether the included file should be printed in the listing of the program. Note, however, that this specification is overridden by the specification of /LIST, /SHOW=INCLUDE, or /SHOW=NOINCLUDE on the EPASCAL command line (see Chapter 16, "Program Development").

Lines and Line Numbers

Source files are divided into lines. A line break (the division between two lines) is a valid separation for lexical tokens and may not occur within a token; that is, a token must be all on one line. The compiler assigns line numbers to the source lines. These appear on the listing and within error messages and may be used to position the debugger to a particular source line.

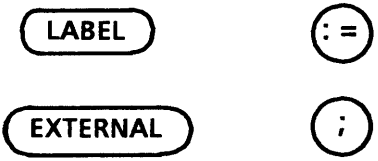
The line numbers assigned by the compiler include numbers for lines obtained from auxiliary source files using %INCLUDE. Hence, they may not correspond to the normal line numbering in the source file (or include file). Compiler error messages also give a source file name and source file line number. (For more information on line numbering in VMS, see the appropriate VMS documentation, including that for the particular editor you are using.)

Syntax Conventions

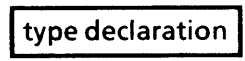
Appendix B contains an alphabetical collection of syntax diagrams representing the syntactic categories of the VAXELN Pascal language. These categories are explained individually throughout this manual, in the appropriate sections.

In general, Pascal syntax diagrams are read from left to right and top to bottom. Arrows point the way in case of ambiguity. The following additional conventions are used:

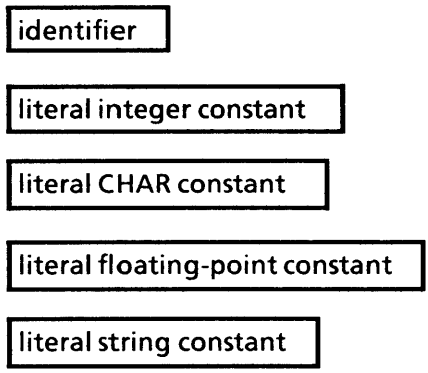
- Rounded symbols (ovals or circles) denote syntax elements that are entered in your program exactly as shown. These elements consist of Pascal reserved words, specific nonreserved identifiers, special symbol operators, or punctuation marks. For example:



- Rectangular symbols denote syntax elements that are, in most cases, syntactic categories described by other syntax diagrams. For example:

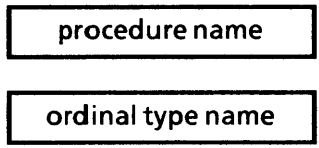


- Rectangular symbols also denote terminal syntax elements defined by lexical rules. The following syntactic categories represent lexical elements:



  The rules for identifiers are given earlier in this chapter. The rules for literal constants are given in Chapter 4, "Constants."

- Syntax elements ending with the words "name" or "type name" denote identifiers that have been declared (or are predeclared) as the appropriate type of name. For example:



Call Format Conventions

Appendix C contains an alphabetical summary of the call formats of the procedures and functions available in VAXELN Pascal. Predeclared functions and procedures must be called as documented. In particular, the documented call formats for kernel procedures use the following conventions, unless the name is prefixed with KER$:

- All arguments shown as undecorated identifiers are *positional*; they must come first in the call, in the order shown. If they are output arguments, they must be variables.
- The argument form NAME := argument means that NAME is the name of a formal parameter, and the argument is optional. These are always the last arguments in the VAXELN Pascal call and can occur in any order.
- An argument name suffixed with "-list" means that a variable number of arguments correspond to a single procedure parameter; in some cases, no argument is necessary.

In other predeclared routines, optional parameters are named in *italics*; however, the nonpositional form cannot be used in these cases.

Chapter 2
Program Structure

Introduction

This section briefly discusses the overall structure of a VAXELN Pascal program, first in its relation to program execution, then in its relation to the VAXELN Pascal compiler and the VAX/VMS linker.

In VAXELN, a program is executed as a *job*. The initial set of jobs in a system (hence the set of programs initially executed) is defined by using the System Builder. Additional jobs may be created by using the CREATE_JOB predeclared procedure (or kernel service) or the VAXELN debugger. More than one job may be executing a program at the same time. In this case, there is no connection between the jobs, except via the methods described in Chapter 12, "Interjob Communication."

The "main" routine of a job is a *PROGRAM block*, whose source text form is PROGRAM ... END. This routine may invoke other routines: procedures and functions as in standard Pascal, but also process blocks and interrupt service routines.

A *process block* is invoked via the CREATE_PROCESS kernel service, rather than the normal procedure-call notation. This creates a new subprocess in the job, which executes in parallel with the job's main process and any other subprocesses. The process block is the main routine of the subprocess; it can, in turn, invoke procedures, functions, and other process blocks.

Interrupt service routines are used in device driver programs. They are invoked asynchronously as the result of the occurrence of a device interrupt. The connection between an interrupt service routine and particular hardware interrupts is made via the CREATE_DEVICE kernel service.

Syntactically, a complete routine definition consists of a heading followed by a routine body. The *heading* specifies the type of routine (for example, PROGRAM, FUNCTION, or PROCEDURE) and its parameters, if any. The *routine body* has the same form for all routines: a set of declarations, followed by a compound statement (BEGIN ... END) giving the executable code. The declarations define constants, types, variables, and additional routines.

In Pascal, each routine body is a block. This means that declarations of constants, types, variables, and additional routines in the routine body are known by name only within the block. Declaration of a name in a block applies only within the block and overrides any declaration of the same name in a containing block.

In VAXELN Pascal, the block structure defined by routine bodies is extended to encompass *outer-level declarations*; that is, declarations not contained within any routine. In this regard, a complete VAXELN program is a set of outer-level declarations, one of these being the declaration of a PROGRAM block.

For purposes of separate compilation, a VAXELN program may be divided into several *compilation units*. One invocation of the compiler compiles one source file (compilation unit), producing one object module. In addition to containing object code for input to the VAX/VMS linker, the object module contains an exported symbol table containing the outer-level declarations to be made available to other compilation units.

For example, if compilation unit B depends on compilation unit A, A is compiled first, then A's object module is included in the compilation of B. This is explained in more detail in Chapter 16, "Program Development."

A compilation unit may consist of a single outer-level routine declaration (that is, a PROGRAM block, PROCEDURE, FUNCTION, or PROCESS_BLOCK declaration), or it may be an explicit module (that is, a series of declarations introduced by the reserved word MODULE and terminated by an END).

A *module* contains a set of outer-level declarations, and it may contain module headers specifying more explicitly the names to be exported by the module, the names exported from other modules to be used by this module, and the names of other modules to be used in the compilation.

Note that every compilation unit is treated as a module and produces a VAX/VMS object module. A complete program may contain modules provided as part of the VAXELN development system in addition to modules explicitly compiled by the user. Programs may also use object modules produced by other language processors. In this case, VAXELN Pascal declarations containing the EXTERNAL attribute or the EXTERNAL directive are used to define how the external data item or routine, respectively, is to be viewed within Pascal.

Figure 2-1 illustrates the modules making up a complete VAXELN Pascal program. The relationship between the PROGRAM block and its associated outer-level declarations is described in the text accompanying the figure.

```
MODULE zmod;
INCLUDE service1, report, filedef;
PROGRAM ZENO(outdata);
    VAR
        slave: PROCESS;
        sisterport: PORT;
    PROCEDURE inform(
        p: PORT);
        BEGIN ... END;
    BEGIN
        CREATE_PROCESS(
        slave,service1);
        WAIT_ANY(slave);
        REPORT('slave done');
        CREATE_JOB(
        sisterport,sister);
        inform(sisterport)
    END;
END; { End of zmod. }
```

```
PROCESS_BLOCK service1;
    VAR ...:
    BEGIN...END;
```

```
PROCEDURE report(
    s: VARYING_STRING(80));
    VAR ...:
    BEGIN...END;
```

```
MODULE filedef
    EXPORT outdata;
    VAR outdata:
        FILE OF RECORD...END;
END; {of filedef}
```

Here, the procedure inform *and the variables* slave *and* sisterport *are declared inside the PROGRAM block (and are available only there).*

The procedure report, *process block* service1, *and module* filedef *are compiled separately. When the module* zmod *is compiled, these object modules are included in the compilation. This establishes declarations of* service1, report, *and* filedef *in the export block (see Figure 2-10).*

Figure 2-1. The Modules Making Up a Complete Program

The module containing PROGRAM ZENO can be compiled by the command

```
EPASCAL ZENO + ZENOMOD/LIBRARY
```

where ZENO.PAS is the source file containing the PROGRAM block and ZENOMOD.OLB is an object library you have built to hold the object modules for procedure report, process block service1, and the declarations in module filedef, which were compiled previously. The compiler produces the object module ZENO.OBJ from the source file.

Notice that in module filedef, the name outdata is explicitly exported, making it available at the outer levels of other modules; usually, it is best to put the declarations of shared data in a separate module (not the PROGRAM block), as shown here. The names of routines, when used as modules, are implicitly exported, as are service1 and report in this case.

After compiling, the object modules can be linked by the command

```
LINK ZENO + ZENOMOD/LIBRARY
```

which produces the program image file ZENO.EXE from the object module ZENO.OBJ and the object modules contained in the library ZENOMOD.OLB. The program image is then ready to be included in a VAXELN system, which you create with the System Builder.

Chapter 16, "Program Development," contains more information on the EPASCAL command. In addition, The VAX/VMS librarian and linker and the VAXELN System Builder are discussed in the *VAXELN User's Guide.*

The remainder of this chapter describes the structure of VAXELN Pascal programs in terms of compilation

units, modules, PROGRAM blocks, and routines bodies. The chapter concludes with a discussion of the scope of declarations.

Compilation Units

An invocation of the VAXELN Pascal compiler compiles one source file, producing one VAX/VMS object module. The text of the source file, expanded by inclusion of any files specified via the %INCLUDE construction (see Chapter 1), must satisfy the syntax for compilation units, as shown in Figure 2-2.

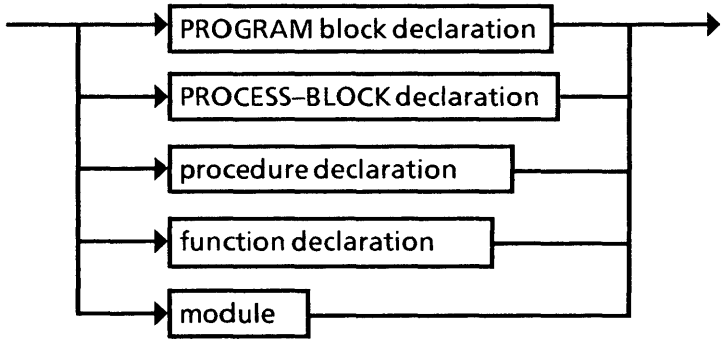


Figure 2-2. Compilation Unit Syntax

The structure of a compilation unit that is an explicit module is explained in the next section. In the other cases, the compilation unit is treated as a module whose exported symbol table contains just the declaration of the PROGRAM block, process block, procedure, or function, and the module's name is the same as the routine's name.

Note that if a complete VAXELN Pascal program is a standard Pascal program, there will be only one compilation unit and it will be a PROGRAM block.

Modules

A module contains a set of outer-level declarations that are compiled as a single compilation unit. The various forms of headers name the module and provide control over the exportation of outer-level declarations from the module and the importation of names from other modules.

In addition to the forms of declaration mentioned earlier in this chapter (that is, constant, type, variable, function, procedure, PROCESS_BLOCK, PROGRAM block, and interrupt service routine declarations), a module may contain *separate routine bodies*. These complete the definitions of functions and procedures declared in other modules, using the SEPARATE directive. (See Chapter 8, "Procedures and Functions," for more information on separate routine bodies.)

The syntax for modules is shown in Figure 2-3.

Module Headers

A *module header* is the first part of a module, excluding any preceding comments. The syntax is shown in Figure 2-4.

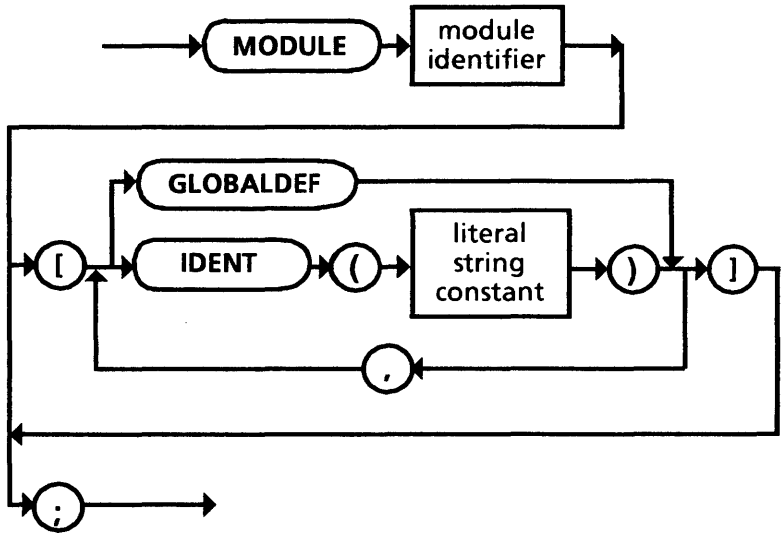


Figure 2-4. Module Header Syntax

The module identifier is established as the name of the module. This does not conflict with any other form of declaration of the identifier.

If GLOBALDEF is specified, the names of all ordinal constants exported from the module are made known to the VAX/VMS linker as global values. (See "Exported Symbols and the Linker," later in this section.)

If IDENT is specified, the literal string constant is stored in the object module's ident field. The length of the constant must be in the range 2..31. If IDENT is not

specified, a string identifying the compiler version is stored in this field. The ident field is in the first record of a VAX/VMS object module. If the module is examined using the ANALYZE/OBJECT command, this field is displayed as the "module version."

Note: If you want several modules to have the same ident (for example, because it's a program version number), use %INCLUDE to include the string constant.

Note that GLOBALDEF and IDENT are, in a sense, attributes of the module, and like other attributes, they are placed in square brackets.

Export Headers

An *export header* specifies names to be explicitly exported from a module; that is, whose declarations will be included in the export symbol table, so they can be used in the compilation of other modules. The syntax is shown in Figure 2-5.

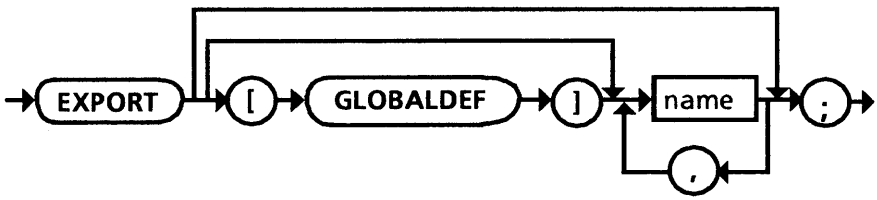


Figure 2-5. Export Header Syntax

If a name is specified in an export header, the name must be declared at the outer level of the module containing the export header.

A module may contain any number of export headers. If it contains none, all outer-level declarations are

exported by default. If any export headers are specified, only names exported by the headers are exported.

Note: An empty export header is allowed. If this is the only export header in a module, no names are exported from the module. The only case in which this is useful is for a module that contains the separate routine body for a routine whose declaration is in another module.

Exporting an enumerated type by name exports all the constants declared by the enumerated type definition.

GLOBALDEF is used in an export header to specify that all ordinal constants exported by that header will be made known to the VAX/VMS linker as global values. (See "Exported Symbols and the Linker," later in this section.)

Note that, in general, a module should not export a name that is the same as a predeclared name. The compiler issues a warning message if this occurs.

Import Headers

An *import header* specifies names to be explicitly imported into the compilation of a module. Here, importing a name means using a name exported from another module included in the compilation. The syntax is shown in Figure 2-6.

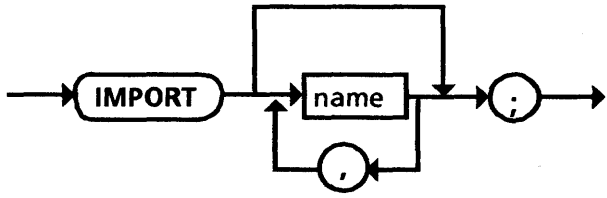


Figure 2-6. Import Header Syntax

A module may contain any number of import headers. If it contains any, the compiler issues a warning message for each imported name not explicitly specified in an import header. In addition, the compiler issues a warning message for any explicitly imported name not actually referenced by the current module. If no import headers are specified, the compiler issues no warnings for use or nonuse of imported names.

Note: An empty import header is allowed. If this is the only import header in the module, the compiler issues warnings for all imported names used in the module.

Include Headers

An *include header* specifies modules to be explicitly included in the compilation of this module. The syntax is shown in Figure 2-7.

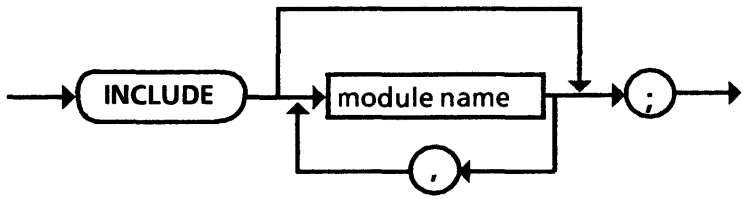


Figure 2-7. Include Header Syntax

Modules can also be included by using the INCLUDE qualifier on the EPASCAL command, by specifying an object module as an input file to the compiler with the MODULE file specification qualifier, and as an indirect result of the compilation of other modules. (See Chapter 16, "Program Development," for a complete discussion of the EPASCAL command and module inclusion.)

Exported Symbols and the Linker

The exported symbol table in a VAXELN Pascal object module is ignored by the VAX/VMS linker. (If you examine the module with the ANALYZE/OBJECT command, the exported symbol table is in the records displayed under the title "IGNORED HEADER (subtype 101)".) However, there is a close relationship between the names exported by a module and the set of global symbols defined by the object module to the linker.

The following exported names are defined as global symbols to the linker:

- The name of a procedure, function, subprocess, program, block, or interrupt service routine, but only from the module that contains its whole body, not just a SEPARATE or EXTERNAL heading.
- The name of a procedure or function whose definition is given by a separate routine body in this module. (The name may actually be exported by a different module.)
- The name of a variable, unless it is declared with the EXTERNAL attribute.
- The name of a string constant.
- The names of ordinal constants governed by an export header with the GLOBALDEF attribute.

Ordinal constants (governed by GLOBALDEF) and variables with the VALUE attribute are made known as values to the linker. String constants and other items are made known as locations. For an exported routine name, this is the location of the routine's entry mask.

The following exported names are *not* made known as global symbols:

- EXTERNAL names (which must be defined as global symbols in a non-VAXELN object module).
- Names of types, routine types, and floating-point constants.
- Names of ordinal constants not governed by GLOBALDEF.

PROGRAM Block

The syntax for a PROGRAM block declaration is shown in Figure 2-8.

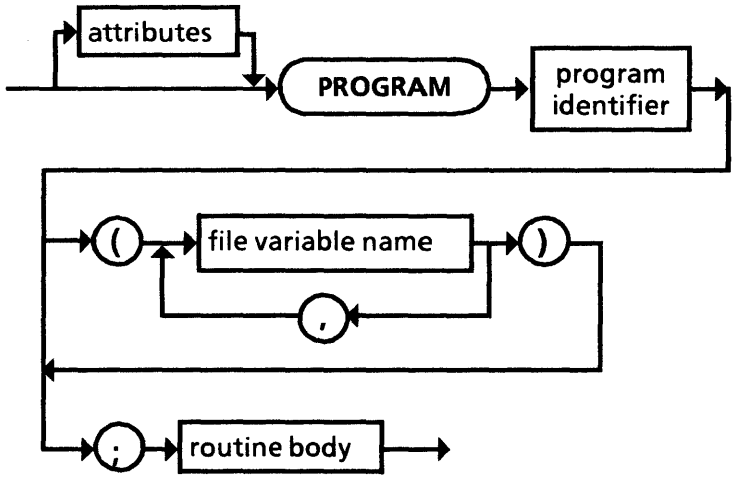


Figure 2-8. PROGRAM Block Declaration Syntax

The program identifier is declared at the outer level as the name of the program block. This declaration is only

allowed at the outer level, and there must be exactly one PROGRAM block declaration in a complete VAXELN Pascal program.

The routine body supplies the local declarations and executable code for the PROGRAM block. One of the attributes UNDERFLOW or NOUNDERFLOW may be specified in the PROGRAM block declaration. If specified, it applies to the routine body, as explained in the section "Routine Bodies," later in this chapter.

The optional file variable names in the PROGRAM block declaration are associated with program arguments, as explained later in this section.

Program Arguments

The arguments of a program are handled differently from those of other routines. Each argument is a string with up to 100 characters. The arguments are specified in the call to CREATE_JOB that creates the job executing the program, or in the *Program Description* menus of the System Builder for jobs started as part of system initialization.

As explained below, a program argument may be associated with a file variable. The value of any program argument (whether or not associated with a file variable) can be obtained via the PROGRAM_ARGUMENT function. The actual number of arguments passed to a program can be obtained via the PROGRAM_ARGUMENT_COUNT function.

Program Files

Specifying a file variable name in the heading of a PROGRAM block associates the corresponding program argument with the file variable. For example:

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

Here, the first program argument is associated with OUTPUT, the second with INPUT.

The argument value is used as the file specification when the file variable is opened, unless it is explicitly opened by a call to the OPEN procedure with a file_name argument.

A file variable name in the PROGRAM heading may be predeclared, declared at the outer level, or declared in the PROGRAM block. If the name is declared in more than one of these blocks, the innermost declaration applies, and it must declare the name as a file variable. In this circumstance, the compiler issues a warning message because there may be a misunderstanding of the effect of specifying the name as a program file.

Program Names

An identifier is declared as a program name by its occurrence as the program identifier in the heading of a PROGRAM block. There is no language construction in which the identifier can be used to specify the program by name. However, a string value specifying the name is used as an argument to kernel services, such as CREATE_JOB.

The PROGRAM block's object module defines the name to the VAX/VMS linker as a global symbol denoting the PROGRAM block's entry point.

Job Activation and Termination

A job is created by the VAXELN System Builder at the time of system initialization, or by using the CREATE_JOB kernel service or the VAXELN debugger. (See the *VAXELN User's Guide* for more information.)

The VAXELN kernel establishes the job's P0 address space, which is shared by all processes in the job, and the P1 address space (stack) for the job's main process (the process in which the PROGRAM block executes). Program arguments are stored in P0 space so they can be accessed by the PROGRAM_ARGUMENT function and by the I/O run-time routines (for file opening).

No files are initially open, but INPUT, OUTPUT, and files named in the PROGRAM block's header may be implicitly opened by the first I/O operation on them. (See Chapter 15, "Input and Output," for more information.)

The kernel then activates the PROGRAM block's routine body (discussed in the next section). After executing any required prolog code, the PROGRAM block's compound statement (BEGIN ... END) is executed.

A job terminates when execution of the PROGRAM block's compound statement completes, or when the job's main process is terminated via the DELETE or EXIT procedures, or by the occurrence of an unhandled exception.

Job termination also terminates all existing subprocesses of the job. Any files not already closed are closed, but this is not a normal close operation and data in buffers may be lost (see Chapter 15).

If the job was created by a call to CREATE_JOB with the NOTIFY parameter specified, a "termination message" is sent to the specified port. (See the description of CREATE_JOB in Chapter 11, "Subprocesses and Synchronization," for more information).

Finally, all the terminated job's resources are returned to the kernel.

Routine Bodies

A routine body supplies the local declarations and code for a routine.

The syntax is shown in Figure 2-9.

The compound statement (BEGIN ... END) contains the routine's executable code. The various declaration categories provide declarations of named constants, types, variables, functions, and procedures that are local to the routine. Note that the declarations can occur in any order.

Explicit declaration of statement labels may be accomplished by using the reserved word LABEL followed by the labels (identifiers or unsigned decimal integers), as shown in the syntax. Labels so declared must occur as statement labels within the routine body's compound statement. Explicitly declaring labels in this way is optional, but if any labels are explicitly declared, all the labels within the compound statement must be explicitly declared.

Routine Body Activation, Stack Frames, and Termination

Except in the case of procedures and functions with the INLINE attribute, invoking a routine creates an activation of its routine body by allocating a *stack frame* and setting hardware register FP to denote this frame. The structure of a basic stack frame is described briefly in the *VAXELN User's Guide* and a more detailed description is contained in the *VAX Architecture Handbook*.

The stack frame contains sufficient information for execution of a hardware RET instruction, which will free the stack frame and properly return control to the invoking routine body (upon procedure or function termination) or to the kernel (upon termination of other routines).

The stack frame of a procedure or function is allocated on the stack (P1 space) of the current process, immediately following the stack frame of the invoking routine body. The sequence of stack frames in a

process's stack defines the current calling chain for procedures and functions and is sometimes referred to as the *call stack*. It starts with the stack frame for the process's main routine (PROGRAM block or process block) and ends with the frame for the routine currently being executed (the frame denoted by register FP).

For PROGRAM blocks and process blocks, the kernel creates the stack frame at the base of the new process's P1 space as part of process creation. The stack frame for an interrupt service routine is allocated on the interrupt stack (see the *VAX Architecture Handbook*).

Execution of a routine body's explicit code begins at the first statement in the routine body's compound statement. In general, this is preceded by the execution of prolog code that extends the stack frame (that is, allocates additional stack space) to hold local variables and code that initializes local variables.

When execution of the compound statement completes, the routine body terminates, and control is returned to the invoking routine or to the kernel, as appropriate. This frees the entire stack frame (that is, all the storage used by the block activation). If the routine body has any local file variables, they are automatically closed before the termination completes.

A routine body activation may also terminate due to execution of a GOTO statement whose target is in a preceding routine body activation on the stack. In this case also, the stack frame is freed and local files are closed. (See Chapter 7, "Pascal Statements," for an explanation of an up-level GOTO statement.)

In addition, if the routine body has an established exception handler at the moment of termination, it is invoked with the exception SS$_UNWIND, as

explained in Chapter 13, "Errors and Exception Handling."

A routine body activation may terminate abnormally due to termination of the containing process via the EXIT or DELETE procedures. In this case, local files are not closed (although they may be closed in another way at job termination), and there is no unwinding of the stack.

UNDERFLOW and NOUNDERFLOW Attributes

The UNDERFLOW and NOUNDERFLOW attributes enable and disable detection of floating-point underflow in a routine's code. As shown in the syntax for the PROGRAM block declaration (Figure 2-8), the attribute is specified at the beginning of the PROGRAM heading, preceding the reserved word PROGRAM.

Similarly, one of these attributes can be specified at the beginning of the routine heading in the syntax for the various other forms of routine declaration; that is, PROCEDURE, FUNCTION, INTERRUPT_SERVICE, or PROCESS_BLOCK declarations. The routine declaration must contain a routine body; it cannot specify a directive, such as EXTERNAL.

Note: UNDERFLOW and NOUNDERFLOW are mutually exclusive and are incompatible with the INLINE attribute.

Underflow detection is disabled by default. An UNDERFLOW or NOUNDERFLOW attribute applies to nested routine bodies unless overridden by use of the complementary attribute.

For a general discussion of the concept of floating-point overflow and underflow, see Chapter 6, "Expressions and Operators."

Scope of Declarations

In this manual, the term *declaration* refers to a definition of an identifier as the name of a data type, data item, or routine. In the syntax diagrams, the occurrence of an identifier in a declarative context is denoted by the category name "identifier," possibly with a modifying adjective indicating the type of declaration; for example, "constant identifier."

The occurrence of an identifier in a context referencing a declared item is denoted by use of the category name "name," possibly with a modifying adjective indicating the type of item; for example, "type name."

More than one declaration of a name is allowed. When the compiler interprets a reference to a name, it must determine which declaration, if any, governs the reference. In most cases, this is determined by the block structure of the compilation unit. The general principles are:

- A block must not contain two declarations of the same name.
- A reference to a name is governed by the declaration in the innermost (most deeply nested) block that contains the reference and declares the name.
- It is an error if the name is not declared in some block containing the reference.

This picture is a bit oversimplified, however. In addition to the blocks explicitly shown in the syntax diagrams (that is, the bodies of routines), there are implicit blocks related to declarations outside of any routine. Also, there are language constructions that declare names with scopes more or less than a block. The following subsections discuss these topics.

Block Structure

Each routine body in a compilation unit is a block. In addition, there are three implicit blocks containing declarations of predeclared names, declarations exported from other modules, and the outer-level declarations of the compilation unit. The nesting of these blocks is shown in Figure 2-10.

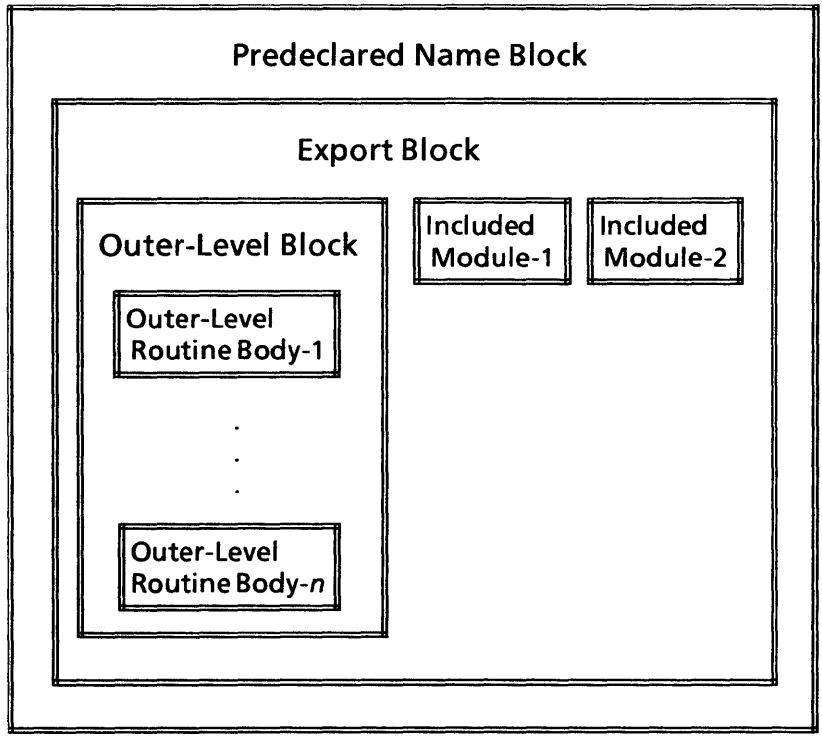


Figure 2-10. Nested Block Structure

The predeclared name block contains all the declarations built into the language; for example, the type BOOLEAN and the constant TRUE.

The export block contains the declarations of names exported from other modules that are included in this compilation. Non-exported declarations within included modules are not visible in this compilation. It is an error if the current compilation unit exports a name that is also exported by another module. (If one thinks in terms of the entire program rather than individual compilation units, then all exported declarations belong to one block, and there must not be any duplicates.)

The outer-level block contains all constant, type, variable, and routine declarations in the compilation unit that do not belong to (that is, are not within) a routine body.

Notion of Scope

For most declarations, the *scope* of an identifier is the block in which it is declared. Duplicate declarations in the same scope are not allowed; however, one scope can be nested inside another. For example:

```
PROCEDURE p;
   VAR a: INTEGER;

      PROCEDURE q;
         CONST a = 3;
         BEGIN...END;

   BEGIN ... END;
```

Here, the scope of the declaration of a as an integer variable is the entire block defined by procedure p. In turn, the scope of the declaration of a as the constant integer value 3 is limited to the nested block defined by procedure q.

Special Declarative Scopes

The VAXELN Pascal declarations whose scope is not exactly a block are summarized below. If the occurrence of a name is not governed by one of these special scopes, the name is resolved according to the general block structure principles given earlier in this section.

Routine Parameters

A routine parameter identifier is declared by its occurrence in the parameter list of a routine heading (excluding PROGRAM block headings, which do not have parameters in the normal sense). The scope of the parameter declaration consists of the parameter list, the function result type (FUNCTION heading), and the routine body (if it exists). Note that if the SEPARATE directive is used, the routine body may be in another module.

Extent Parameters

An extent parameter identifier is declared by its occurrence in a flexible type definition. The scope of the declaration is the type on the right-hand side of the "=" in the flexible type definition (See Chapter 3, "Data Types," for more information on flexible types.)

Field Names

A field identifier is declared by its occurrence in the field list of a record type definition. This declaration has no scope in the ordinary sense. (The field name can be used in a field reference that specifies the containing record.) However, a WITH statement can be used to establish the field name locally.

Names Established by the WITH Statement

A WITH statement can be used to establish field names or names specified using "WITH identifier AS" as data item names in the body of the WITH statement. (See Chapter 7, "Pascal Statenments," for more information on the WITH statement.)

Module Names

An identifier is established as a module name by its occurrence following the reserved word MODULE in a module header, or if it is the name of a routine whose declaration occurs as a complete compilation unit. This establishment of a name has no scope in the ordinary sense, and it does not conflict with any other declaration of the name, except as a module name.

Order of Declarations, Circularity

Within a block, declarations can occur in any order. A declaration may depend on one that follows it, such as:

```
VAR a: t;

TYPE t = 0..127;
```

With one exception, a declaration must not be circular; that is, it is not allowed to depend on itself directly or through another declaration. Therefore, the following declaration is not allowed:

```
TYPE t1 = RECORD    { Circular declarations. }
   x: t2
   END;

TYPE t2 = ARRAY[1..10] OF t1;
```

The one exception to the rule against circularity is the use of a pointer type of the form " ↑ some-type", where "some-type" is the name of a non-flexible type. (See Chapter 3, "Data Types," for more information on pointer types.) The following declaration is valid:

```
TYPE some-type = RECORD
   x: STRING(10);
   link: ↑some-type;
   END;
```

Note that this does not allow circularity involving a pointer to a bound flexible type (see Chapter 3). The following is incorrect:

```
TYPE t(n: INTEGER) = RECORD
   x: STRING(n);
   link: ↑t(n)        { Circular declaration. }
   END;
```

To manipulate a structure of this sort, you must use ↑ ANYTYPE (see Chapter 3). For example:

```
TYPE s(n: INTEGER) = RECORD
   x: STRING(n);
   link: ↑ANYTYPE
   END;

VAR p: ↑s(10);
   p := p↑.link     { Advances to next record in list. }
```

Chapter 3
Data Types

Every Pascal data item is associated with a data type. The *type* of a data item defines the kind of values it can have and the operations that can be performed on it. In addition, the type determines the item's representation.

This chapter discusses the declaration of data type names, the definition of each of the various VAXELN Pascal data types, the notion of type equivalence, and the rules for internal representation of data, including the use of attributes to modify the normal representations.

Type Declarations

In most cases, the type of a Pascal data item is specified by using the name of a data type, either one of the VAXELN predeclared types or a type introduced by a type declaration. The syntax of type declarations is shown in Figures 3-1 through 3-3. Note that these figures also show the various possibilities for referring to a type or introducing a new type.

In Figure 3-1, the type identifier is declared as a type. If the type on the right-hand side of the equal sign is simply a type name (see Figure 3-3), the identifier is declared to be a synonym for the type denoted by the type name. In all other cases, the identifier is declared as the name of a new type given by the type definition on the right-hand side of the equal sign.

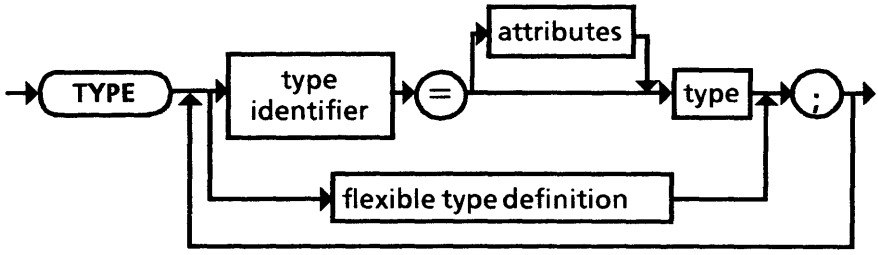


Figure 3-1. Type Declaration Syntax

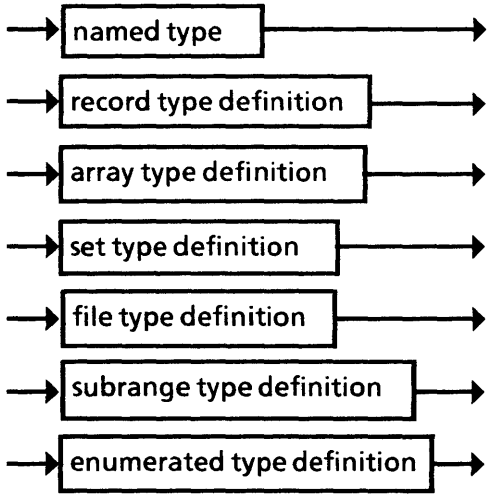


Figure 3-2. Type Syntax

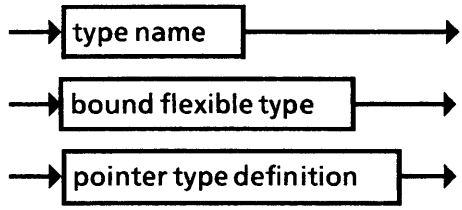


Figure 3-3. Named Type Syntax

As shown in Figure 3-1, attributes may be specified to modify the normal representation of certain data types; their use is explained in the last section of this chapter. A flexible type definition declares an identifier as the name of a new flexible type; this is explained in the section "Flexible Types," later in this chapter.

Ordinal Types

Ordinal types consist of the type INTEGER and certain other types whose values have a one-to-one correspondence with a consecutive set of integers. These values are ordered so that each has a unique ordinal value that indicates its position in a list of all the values of the type. Ordinal types are used as extent parameters in flexible data types, as indices in arrays, as type names in variant records, as base types in sets, and as operands in extent expressions.

The representation of ordinal types can be modified by packing; that is, by being an element of a PACKED record or array. In addition, the representation can be explicitly controlled by the attributes BIT, BYTE, WORD, or LONG, as discussed later in this chapter.

VAXELN Pascal supplies predefined ordinal types for integer, character, and Boolean data. In addition, Pascal allows you to define your own ordinal types in one of two ways:

- By enumerating each value of the type (enumerated types).
- By defining the type as a subrange of another ordinal type (subrange types).

INTEGER Data Type

The type INTEGER represents whole numbers in the range -2^{31} through $2^{31}-1$; that is, $-2{,}147{,}483{,}648$ through 2,147,483,647. (For representation of numbers of larger magnitude, see "LARGE_INTEGER Data Type," later in this chapter.)

Internal Representation of INTEGER Data

The normal representation of INTEGER data is as a byte-aligned VAX longword (32 bits, or 4 bytes) representing a signed integer in 2's-complement form.

Subranges of the INTEGER type are packable data types, although INTEGER itself is not. When a packable integer item is immediately contained in a PACKED record or array, its boundary requirement is only bit alignment, and it occupies only *n* bits, where *n* is generally less than 32. When the subrange contains only nonnegative values, *n* is the minimum number of bits required to represent the subrange's maximum value as an unsigned integer. Otherwise, *n* is the minimum number of bits required to represent the low and high subrange values as signed (2's-complement) integers.

Packable data and *boundary requirement* are discussed in "Data Representation," later in this chapter.

INTEGER data items can have their representations specified exactly by the BIT, BYTE, WORD, or LONG attribute, which overrides the packing of INTEGER subranges described in the preceding paragraph.

CHAR Data Type

The data type CHAR represents single characters. The set of valid characters includes uppercase and lowercase letters, the digits 0–9, and an assortment of punctuation marks and nonprinting characters, as shown in Table 3-1.

The Character Set

CHAR is an ordinal type; each character in the set corresponds to one of a series of integers (the ordinal values of the character set) that start at zero. The ordinal value of a character can be obtained with the ORD function; the character corresponding to an ordinal value can be obtained with the CHR function.

The VAXELN Pascal character set is compatible with the American Standard Code for Information Interchange (ASCII). The ASCII set specifies 128 characters and provides for an additional, unspecified set of 128.

The rules and relationships for this character set are as follows:

- The *relationship* between two characters is the same as the relationship between their ordinal values. That is, char1 < char2 if and only if ORD(char1) < ORD(char2).
- The *numeric characters* (0, 1, and so on) are numerically ordered and also are contiguous; that is, '1' < '2' is TRUE. Contiguity means that if ORD('0') is *n*, then ORD('1') must be $n+1$, and so forth.
- The numeric characters (0–9) all have ordinal values less than those of the alphabetic characters (A–Z and a–z).
- The *alphabetic characters* (both uppercase and lowercase) are alphabetically ordered, so 'A' < 'B' is TRUE.
- The *uppercase letters* (A–Z) have ordinal values that are less than those of the *lowercase letters* (a–z). Furthermore, the ordinal values of uppercase and lowercase versions of the same

letter always differ by 32. For example, ORD('a') – ORD('A') = 32.

- The *printable characters* are those with ordinal values greater than or equal to 32 and less than 127. (CHR(32) is the space character.) Characters with values less than 32 do not have a standard printable representation, although some, such as line feed (CHR(10)) and carriage return (CHR(13)) are often used to format printed documents.

Internal Representation of CHAR Data

The normal representation of CHAR data is as an eight-bit byte containing ORD(charvalue) as an unsigned integer.

Subranges of CHAR are packable data types, although CHAR by itself is not. When a packable CHAR data item is immediately contained in a PACKED record or array, its boundary requirement is only bit alignment, and it occupies *n* bits, where *n* is the minimum number of bits required to represent the item's maximum value as an unsigned integer.

CHAR data items can have their representations specified exactly by the BIT, BYTE, WORD, or LONG attribute, which overrides the packing of CHAR subranges described in the preceding paragraph.

BOOLEAN Data Type

The data type BOOLEAN represents the results of relational operations (for example, A < B) and logical operations (for example, F OR G). It consists of the two constant values TRUE and FALSE. (BOOLEAN values are represented by these identifiers in programs and in textfiles).

BOOLEAN is an ordinal type, with FALSE and TRUE having the ordinal values 0 and 1, respectively.

Internal Representation of BOOLEAN Values

The normal representation of BOOLEAN data is as an eight-bit byte. The value TRUE is encoded as the low-order bit set and the rest zero. FALSE is encoded as all zeros. The seven high-order bits are ignored when BOOLEAN values are evaluated.

BOOLEAN and its subranges are packable data types. When a BOOLEAN data item is immediately contained in a PACKED record or array, its boundary requirement is only bit alignment, and it occupies only one bit. Thus, a PACKED ARRAY [1..*n*] OF BOOLEAN occupies only *n* bits of storage.

BOOLEAN data items can have their representations specified exactly by the BIT, BYTE, WORD, or LONG attribute, which overrides the packing of BOOLEAN values described in the preceding paragraph.

Enumerated Types

An enumerated type has a finite set of named values (at most 32,767 values) introduced in the enumerated type's definition.

An enumerated type definition has the form shown in Figure 3-4.

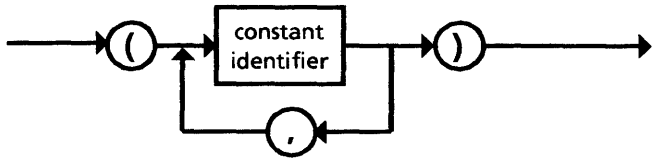


Figure 3-4. Enumerated Type Definition

This defines a new enumerated type and *declares* each constant identifier as the name of one of the type's distinct ordinal values. If the constant identifiers in the definition are x_1, ... x_n, in left to right order, then $ORD(x_i) = i-1$.

For example,

```
TYPE
   season = (spring, summer, autumn, winter);
```

declares the enumerated type season and the four constants spring, summer, autumn, and winter. ORD(spring) = 0 and ORD(winter) = 3.

Internal Representation of Enumerated Data

Enumerated data items normally are byte-aligned quantities occupying either a byte (eight bits) or word (16 bits), depending on the total number of declared values. They are stored in a byte if they have no more than 256 possible values. The data item contains the ordinal number of its current value. Thus, if the ordinal number of the maximum value is at least 128 but less than 256, the representation is an unsigned byte.

Enumerated types and their subranges are packable data types. When an enumerated data item is immediately contained in a PACKED record or array, its boundary requirement is only bit alignment, and it occupies *n* bits, where *n* is the minimum number of bits needed to represent the item's maximum value as an unsigned integer.

Enumerated data items can have their representations specified exactly by the BIT, BYTE, WORD, or LONG attribute, which overrides the packing of enumerated items described in the preceding paragraph.

For example, the variable e, declared as

```
VAR e: (red, yellow,blue);
```

occupies one byte and is byte-aligned. However, if the same type defines a field in a PACKED record, the field occupies only two bits because ORD(blue) is 2, which can be represented in two bits.

Subrange Types

Subrange types denote a subrange of values of an ordinal type. They are themselves ordinal and can be specified wherever ordinal types are valid; for example, as the index range of an array. Subranges are useful for documenting the actual range of permissible values for a variable or type.

The definition of a subrange type specifies the minimum and maximum values, which must be constants of the same ordinal type.

Normally, a variable of a subrange type requires the same amount of storage as its host type. For example, a subrange of integers requires 32 bits. However, if the subrange data is a component of a PACKED array or record, it has a packed representation, as explained under the particular ordinal data type.

A subrange item can have its representation specified exactly by the BIT, BYTE, WORD, or LONG attribute, which overrides the packing described in the preceding paragraph.

It is a range violation if a value outside the specified range is assigned explicitly to a subrange variable. Referring to a subrange variable when its value is out of range (for example, because it is uninitialized or has an out-of-range value due to typecasting) is an unpredictable error.

A subrange type definition has the form shown in Figure 3-5.

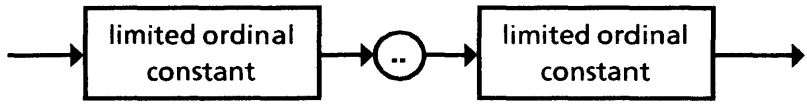


Figure 3-5. Subrange Type Definition

Each of the limited ordinal constants in the subrange type definition is of the same ordinal type. An ordinal constant value must be specified in this context; a general extent expression is not permitted. The constant on the left supplies the lower bound (minimum value) of the subrange and must be less than or equal to the constant on the right. If the two constants are equal, the subrange denotes that single value.

A typical declaration of a subrange-of-INTEGER variable is:

```
VAR sub : 0..255;
```

Here, the variable sub can have only those values in the designated subrange, 0–255.

Thus:

```
sub : = 128;  { Valid assignment. }
sub : = 256;  { Range violation. }
```

Set Types

In Pascal, a set value is a set of ordinal values, all of the same basic ordinal type, which is called the set's *element type*. If the element type is INTEGER, the values must be in the range 0–32,766.

A set type is defined by using the reserved words SET OF and an ordinal type. The ordinal type determines both the element type of the set and the minimum and maximum values that may be contained in sets of the type. A set type may be defined as PACKED. This can affect the set type's internal representation if it is a small set type; that is, if ORD(maximum-element) ≤ 31. (See "Internal Representation of Sets," later in this section.)

Set Type Definitions

A set type definition has the form shown in Figure 3-6.

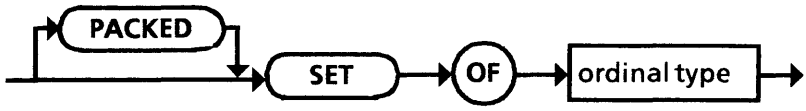


Figure 3-6. Set Type Definition

The basic ordinal type specified in the set type definition is the set type's element type. The minimum and maximum values contained in sets of the type are the minimum and maximum values of the ordinal type. If the element type is INTEGER, the values must be greater than or equal to 0 and must not exceed 32,766. Hence, 'INTEGER' itself is not acceptable here as the ordinal type. Note that the restrictions on the minimum and maximum values are automatically satisfied for all non-integer ordinal types.

The following examples illustrate set type definitions:

```
VAR tset: SET OF CHAR;
    { tset can contain any characters from the
    character set. }
```

```
TYPE subset = SET OF 'A'..'Z';
  { Variables of type subset can contain only the
  uppercase letters A-Z. }
VAR enumset: SET OF (red,orange,yellow);
  { enumset can contain only the enumerated values
  red, orange, and yellow. }
```

Note that set operations require only that the sets involved have the same element type, this always being a basic ordinal type, not a subrange. Hence, with the above examples you can have:

```
VAR s: subset;
  tset : = s + ['a'..'z'];
    { The set construction denotes the set of
    lowercase letters. See Chapter 6 for more
    information. }
```

Internal Representation of Sets

Normally, a set type is represented as a byte-aligned sequence of n consecutive longwords in storage, where n is the minimum positive integer such that ORD(maximum-element) $< 32 \times n$. The bits in these longwords can be numbered 0, 1, ... $32 \times n - 1$, for example, the usual 0 to 31 for a one-longword set.

Bit k is 1 if and only if the element with ORD(element) $= k$ is in the set; otherwise, it is 0. Thus, all bits in the set's storage are defined, including those that do not correspond to the ordinal value of an element allowed by the set's type.

For example, SET OF 0..100 is a sequence of four longwords because that is the minimum number of longwords that has at least 101 bits. The elements are represented by the first 101 bits.

All sets have a (possibly unused) bit corresponding to an element at ordinal 0, and possibly unused bits

corresponding to the ordinal numbers between 0 and the ordinal number of the set's minimum element; any unused bits are always zero. For example, SET OF 2..4 is represented by a sequence of five bits (normally, the five low-order bits of a single longword):

```
Element: 4 3 2
Bit:     4 3 2 1 0
```

Here, bit 3 is 1 if 3 is in the set. (Note that operations that modify bits 0 and 1 are range violations.)

Packed Sets

A set type is packable if it was declared with the word PACKED and is small (ORD(maximum-element) ≤ 31, so that its normal representation would be one longword). When a packable set data item is immediately contained in a PACKED record or array, its boundary requirement is only bit alignment and it occupies only *n* bits, where *n* = ORD(maximum-element) + 1.

Small set items (ORD(maximum-element) ≤ 31) can have their representations specified exactly by the BIT, BYTE, WORD, or LONG attribute, which overrides the packing described in the preceding paragraph.

Floating-Point Types

The predefined data types REAL and DOUBLE provide explicit single- and double-precision floating-point numbers. Floating-point constants use literal decimal notation (for example, 3.2 or 32e – 1), but the internal representation is VAX F, D, or G_floating point. There is no special notation to distinguish REAL or DOUBLE constants; a floating-point constant is converted to its internal representation as REAL or DOUBLE, depending on the context.

REAL Data Type

The data type REAL represents real numbers in the approximate range 0.29×10^{-38} to 1.7×10^{38}. The maximum number of significant fractional digits of a REAL data item, its *precision*, is approximately seven. Numbers with this precision are called *single-precision* real numbers.

Internal Representation of REAL Data

REAL data is represented in the VAX F_floating binary format, which occupies a single byte-aligned longword (32 bits), as shown in Figure 3-7.

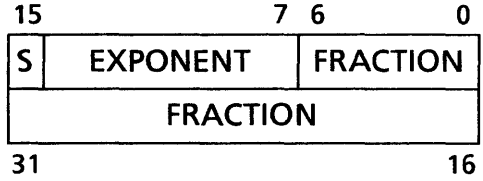


Figure 3-7. Internal Representation of REAL

The format uses a sign bit, an eight-bit exponent (power of two), and a 24-bit fraction.

The zero exponent and zero sign bit together represent the value 0. If the sign bit is 1 and the exponent is zero, the floating-point value is a reserved operand and causes an exception (reserved operand fault). Otherwise, the format represents a normalized floating-point number, in which the fraction's most significant bit is known to be 1 and is not represented explicitly.

The value represented is thus:

$2^{(\text{exponent} - 128)} \times \text{fraction}$

If the sign bit is 1, the value is negative.

DOUBLE Data Type

The data type DOUBLE represents *double-precision* real numbers. Variables of this type are declared with DOUBLE in place of REAL. DOUBLE data can represent numbers with approximately 15 decimal digits.

All operations defined for REAL data also are defined for DOUBLE data. If REAL and DOUBLE operands are used in an expression, the computation is performed in double precision. Via compiler qualifiers, you have the option of using either the VAX D_floating format or the G_floating format for DOUBLE data.

DOUBLE data items in G format have the approximate range 0.56×10^{-308} to 0.9×10^{308}. They have approximately twice the fractional precision of REAL data, and much greater maximum magnitudes.

When the D format is used, DOUBLE data items have the approximate range 0.29×10^{-38} to 1.7×10^{38}, the same as REAL data. In this case, the REAL and DOUBLE types differ only in the amount of fractional precision, which is still about twice the precision for DOUBLE as for REAL.

Note that neither REAL nor DOUBLE data can be assigned to integer variables without use of a conversion function.

Internal Representation of DOUBLE Data

DOUBLE data is represented in the VAX D_floating or VAX G_floating binary format, each of which occupies a byte-aligned quadword (64 bits), as shown in Figures

3-8 and 3-9. In either format, data is represented as a normalized floating-point number, in which the high-order bit of the fraction is known to be 1 and is not explicitly represented.

The value represented is thus:

$$2^{(\text{exponent} - 128)} \times \text{fraction}$$

If the sign bit is 1, the value is negative. The zero exponent and a zero sign bit together indicate the value 0. If the sign bit is 1 and the exponent is zero, the number is a reserved operand and causes a reserved operand fault.

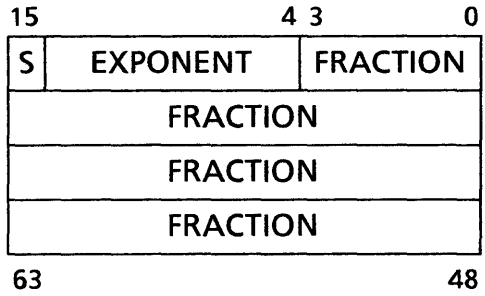


Figure 3-8. G_Floating Representation

VAX G_Floating Format. The G_floating format represents a double-precision floating-point number in 64 bits, offering larger magnitudes than F_floating (single-precision, REAL) numbers and approximately twice the precision. The format uses one sign bit, an 11-bit exponent for a range of approximately $\pm 10^{308}$, and a 53-bit magnitude for about 16.0 decimal digits of accuracy.

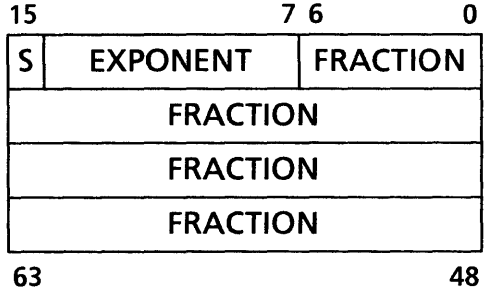


Figure 3-9. D_Floating Representation

VAX D_Floating Format. The D_floating format represents a double-precision floating-point number in 64 bits, offering the same magnitudes as F_floating (single-precision, REAL) numbers but approximately twice the precision. The format uses one sign bit, an 8-bit exponent for a range of approximately $\pm 10^{38}$, and a 56-bit magnitude for about 16.9 decimal digits of accuracy.

Flexible Types

A flexible type is a type with *extent parameters* that determine string lengths, BYTE_DATA item lengths, or array bounds within the data item described by the type. There are three predeclared flexible types: STRING, VARYING_STRING, and BYTE_DATA. These types are discussed individually later in this chapter. Other flexible types can be introduced by declaration.

By itself, a flexible type is not a complete description of data. It can only be used by specifying values for the extent parameters. The combination of a flexible type and extent values is called a *bound flexible type*; for example, STRING(100) describes character strings of length 100.

The extent values in a bound flexible type are specified by *extent expressions*, which are a subset of ordinal-valued expressions. When used in bound flexible types as array bounds, extent expressions may have constant or variable operands. With constant operands, they may also be used in most places where standard Pascal requires a constant.

The following sections cover flexible type definitions, bound flexible types, and extent expressions.

Flexible Type Definitions

The syntax for a flexible type definition is shown in Figure 3-10.

A flexible type definition declares an identifier as the name of a new flexible type. Within the parentheses, identifiers are declared as extent parameters of the flexible type; the scope of these declarations is limited to the flexible type definition. The data type of each extent parameter must be an ordinal type; for example, INTEGER or 0..32767.

The new flexible type is limited to one of the four type definitions shown on the right-hand side of the equal sign in Figure 3-9. The new flexible type can be a pointer to a bound flexible type, it can be defined in terms of another flexible type, or it can be a record type or array type. (Pointer types, record types, and array types are explained in their own sections, later in this chapter.)

Attributes may be specified to modify the normal representation of an array type or record type; their use is explained in the last section of this chapter.

Bound Flexible Types

To use a flexible type, you specify the type name and values for each extent parameter. This combination of flexible type and extent values is a bound flexible type. The syntax is shown in Figure 3-11.

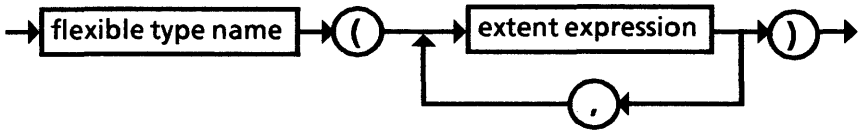


Figure 3-11. Bound Flexible Type Syntax

In a bound flexible type, each extent expression supplies the value of an extent parameter of the type being bound. These extent values match the flexible type's extent parameters left to right and must equal them in number. The extent values must have ordinal types that are assignment compatible with the extent parameters'. It is a range violation if an extent value is outside the range of the corresponding extent parameter or if substitution of the value into the governing flexible type definition leads to a range violation.

Examples

Consider the following flexible type declaration:

```
TYPE vector(n: INTEGER) = ARRAY [1..n] OF REAL;
```

Substituting the extent values into the type's definition gives the basic Pascal interpretation of the bound type. Thus, vector(10) is equivalent to ARRAY[1..10] OF REAL.

A flexible type name cannot be used by itself to specify an item's type; it must be bound. For example, type

```
VAR x: vector;
```

is not allowed because the vector's size is not specified.

Flexible types can be defined in terms of other flexible types, either predeclared or user-declared.

For example:

```
TYPE
   vectorpair(m: INTEGER) =
     RECORD
     a,b : vector(m);
     END;
```

With this declaration, vectorpair(10) is equivalent to:

```
RECORD
a,b : ARRAY [1..10] OF REAL;
END;
```

Pointer types can also be flexible, as in:

```
TYPE
   stringptr(m: 0..32767) = ↑STRING(m);
VAR
   p : stringptr(100);
```

Here, the type of p is equivalent to ↑STRING(100).

In general, the extent values in a bound flexible type are specified by extent expressions. The value of an extent expression within a bound flexible type need not be a constant. It can depend on a routine's value parameters or other values known at entry to a routine. This is the general mechanism in VAXELN Pascal for manipulating dynamically sized data. For example:

```
FUNCTION f(n: INTEGER) : vector(2*n);
```

Here, f is a function returning vectors of length 2*n.

Extent expressions within the type definition can also depend on the flexible type's parameters, as in:

```
TYPE doublestring (n : INTEGER) = STRING(2*n);
```

For routines, the most important feature for describing dynamically sized data is the conformant parameter. Here, the extents of a parameter are derived from the actual argument passed to that parameter. For example:

```
FUNCTION sum( v: vector(<n>) ) : REAL;
   VAR
     k: INTEGER;
     s : REAL : = 0;
   BEGIN
```

```
    FOR k := 1 to n DO s := s + v[k];
    sum := s;
  END;
```

Here, function sum returns the sum of elements in the parameter v. (See Chapter 8, "Procedures and Functions," for more information on conformant parameters.)

Flexible types are normally used within declarations to describe variables, parameters, record fields, and so forth. However, they can also be used in the executable part of a block, in typecast variable references. This gives a completely dynamic description of data in the sense that extents are determined at the moment the variable reference is executed. For example,

```
p↑::STRING(n)
```

means a string whose length is the current value of n and whose location is given by the current value of the pointer p. (See Chapter 5, "Variables," for more information on typecast variable references.)

Extent Expressions

An extent expression is a type of ordinal-valued expression that is used to specify an extent value in an array type or bound flexible type. Extent expressions with constant operands may also be used in several contexts where standard Pascal requires an ordinal constant. Extent expressions denoting true constant values can be used in the following contexts and are evaluated by the compiler:

- On the right-hand side of CONST declarations (CONST i = 5+1).
- In initializers for variables and value parameters (VAR i : INTEGER := 5+1).

- As the upper or lower bound of an array's index range (VAR a: ARRAY[0..5+1] OF INTEGER).
- As extents in bound flexible types (STRING(5+7)).

In the last two cases, extent expressions denoting non-constant values can be used; in these cases, the expression denotes a value that will be determined at entry to a routine or by flexible type binding. For example:

```
TYPE
   flex(m: INTEGER) = ARRAY[0..m - 1] OF REAL;
   { A flexible type. }

VAR
   a1 : flex(10); { The value of m is determined now.}

PROCEDURE p(s : STRING(<n>);
   VAR
     a : PACKED ARRAY[0..n - 1] OF CHAR;
   { The value of n is the length of argument s,
   determined when procedure p is called. }
```

The following rules define the class of extent expressions:

1. Only a limited set of operations is permitted, to ensure that the result is ordinal and can easily be evaluated at compile time (constant operands):
   - The dyadic +, −, *, AND, OR, DIV, and MOD operators (string concatenations with + are not allowed).
   - The monadic operators +, −, and NOT.
   - Relational operators (for example, <).
   - The functions ODD, ORD, PRED, SQR, ABS, CHR, SUCC, and XOR.

2. The terminal operands of the expression must denote ordinal values.
3. Within the definition of a flexible type, a terminal operand of an extent expression can be the name of one of the type's extent parameters (as with type flex in the previous example).
4. Within the type specified for a routine's parameters or function result, a terminal operand of an extent expression can be the name of a conformant extent or value parameter of the same routine. For example, function f returns a string one character longer than its argument:

```
FUNCTION f (STRING(<n>)) : STRING(n + 1);
```

5. Within a type specified in a declaration that is inside a routine, a terminal operand of an extent expression can be the name of a value known at entry to that routine (as with procedure p in the above example). That is, it can be the name of a variable declared outside the routine, or it can be the name of one of the routine's conformant extents or value parameters.

   In cases where it can matter, the compiler generates code to capture the value at entry to the routine. Thus, assignment to the variable within the routine's body does not affect the meaning of the extent expression within the declaration.
6. Except as allowed in 3, 4, and 5 above, a terminal operand of an extent expression must be a literal ordinal constant or the name of an ordinal constant.

Note that the dynamic specification of values allowed by rule 5 can be combined with rule 3 or 4 and with true constants, which is occasionally useful.

For example:

```
TYPE
   matrix (m,n: INTEGER) = ARRAY [1..m, 1..n] OF
   REAL;
   vector (m: INTEGER) = ARRAY [1..m] OF REAL;
CONST
   row = 10;
PROCEDURE p(
   VAR a : matrix(row,col);
   col : INTEGER
   );
   TYPE
     t (alpha : INTEGER) = matrix(2*alpha,col - 1);
```

In the declaration of parameter a, row is a named constant, while col is a value parameter of the same routine, as allowed by rule 4. In the declaration of the flexible type t (within p), 2*alpha uses the extent parameter alpha in accordance with rule 3, and col – 1 uses the value col, known at block entry in accordance with rule 5.

String Types

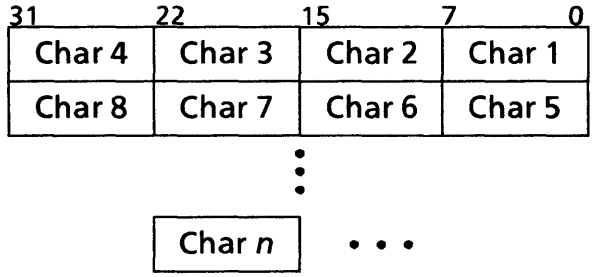
Character string values, or simply, *strings*, are sequences of zero to 32,767 characters. These are the values of the data types STRING and VARYING_STRING. Strings also are the results of *string expressions*, which include certain predeclared functions and the concatenation of two or more strings. In string expressions, variables of types PACKED ARRAY[1..*n*] OF CHAR (when *n* does not exceed 32,767) and CHAR also yield string values.

STRING Data Type

The data type STRING(*n*) represents character strings containing exactly *n* characters ($0 \leq n \leq 32{,}767$).

Internal Representation of STRING Data

A STRING data item is stored as a number of bytes equal to the maximum length *n*, as illustrated in Figure 3-12.



Note that in this memory diagram (as in a typical storage dump), the string characters appear in reverse order on each line.

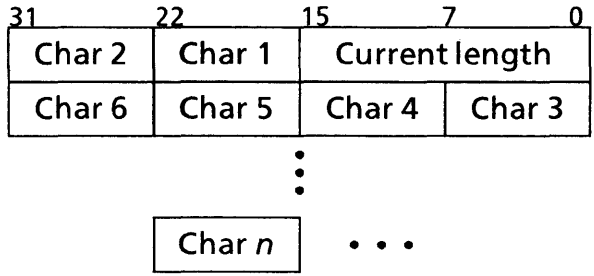
Figure 3-12. STRING(*n*) Representation

VARYING_STRING Data Type

The data type VARYING_STRING(*n*) represents character strings of various lengths, up to a maximum of *n* characters ($0 \leq n \leq 32{,}767$). The current length of a VARYING_STRING variable can be obtained with the LENGTH function.

Internal Representation of VARYING_STRING Data

A VARYING_STRING data item is stored as a 16-bit word containing the current length in characters, followed by a number of bytes equal to the maximum length, as illustrated in Figure 3-13.



Note that in this memory diagram (as in a typical storage dump), the string characters appear in reverse order on each line.

Figure 3-13. VARYING_STRING(*n*) Representation

PACKED ARRAY OF CHAR

For compatibility with other Pascal implementations, PACKED ARRAY[1..*n*] OF CHAR can be used, in most cases, as if it were STRING(*n*). However, there are restrictions on assignments and relational operations, since this is formally an array type. Also, when such an array is used as a string type, it is a range violation if *n* exceeds 32,767.

Strings and the Type CHAR

There is a close relationship between strings and the data type CHAR. In any context that requires a string value, a value of type CHAR is acceptable; it is treated as a string of length 1. Similarly, in any context that clearly requires a value of type CHAR, a string of length 1 is allowed. (When a dynamically sized string value is used in such a context, it is a range violation if its length is not 1.)

Array Types

Array types represent aggregates of elements, all of which have the same type. Through the use of flexible types, arrays can be declared with dynamic extents; that is, with varying numbers of elements in some or all of their dimensions.

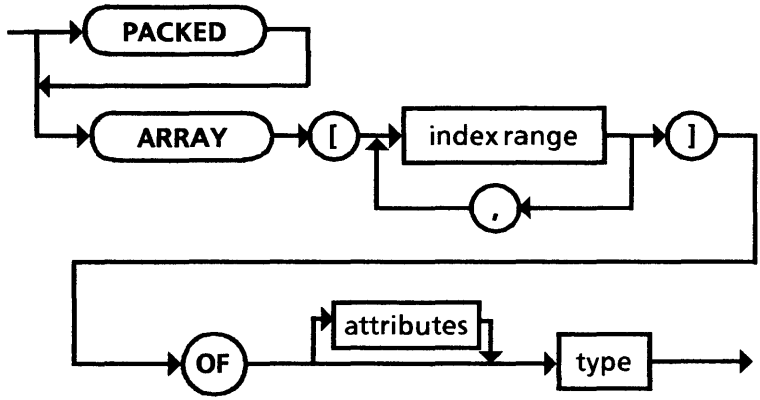
An array type must not specify more than eight dimensions; if its elements are arrays, their dimensions count toward this total. However, it's permissible to have additional arrays of up to eight dimensions within record elements.

Array Type Definitions

The definition of an array type supplies the data type of its elements, the number of its dimensions, and the data types (including minimum and maximum values) of its subscripts, or indices.

The information about the indices in each dimension is expressed by an *index range*. The indices need not have the same data types nor the same minimum and maximum values in each dimension. The syntax for an array type definition is shown in Figure 3-14.

Array Type



Index Range

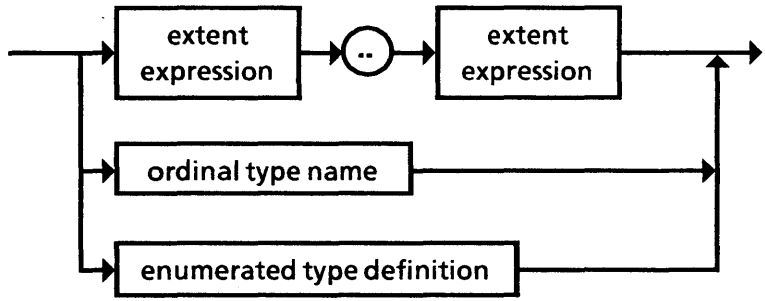


Figure 3-14. Array Type Definition

An array type can be defined as PACKED to specify the most compact storage possible. (More details on PACKED arrays are given later in this section.)

The index range supplies the data type of the array's indices in a dimension, along with the minimum and maximum indices. The data type must be ordinal. The index range can be specified as a pair of extent expressions separated by the symbol '..', the name of an ordinal type, or (rarely) the definition of an enumerated type. The minimum and maximum indices are either the minimum and maximum values of the specified ordinal type, the minimum and maximum values of the enumerated type, or the left and right extent expressions.

The ordinal value of an array dimension's upper bound must be greater than or equal to the ordinal value of (lower-bound − 1). The number of elements in a dimension is (upper-bound − lower-bound + 1). Thus, arrays with zero elements are allowed.

The data type of the array's elements can be of any type, so you can declare arrays of types such as REAL and INTEGER, as well as arrays of arrays, arrays of records, arrays of files, and so forth. Note again that the total number of dimensions in an array type must not exceed eight.

Multidimensional arrays can be denoted either as, for example:

```
ARRAY[range1] OF ARRAY[range2] OF CHAR
```

or as:

```
ARRAY[range1,range2] OF CHAR
```

Attributes may be specified to control the internal representation of an array type; their use is explained in the last section of this chapter.

Declaration of Arrays with Varying Extents

Perhaps the most straightforward use of flexible types is in the declaration of arrays with varying extents. In the following example, a type matrix is defined as a two-dimensional array whose extents are specified in the declarations of variables, permitting matrices of various sizes:

```
TYPE
   matrix(row,column: INTEGER) =
   ARRAY[1..row,1..column] OF REAL;
   { Matrix of reals. }

VAR
   m2x3: matrix(2,3); { A 2 X 3 matrix.}
   m50x10: matrix(50,10); { A 50 X 10 matrix. }
```

A flexible type such as matrix can also be used as part of another flexible type's definition. Here, for example, a record is defined, one field of which is of type matrix:

```
PROGRAM flextypes(OUTPUT);

TYPE
   matrix(row,column: INTEGER) =
   ARRAY[1..row,1..column] OF REAL;
   { Matrix of reals. }
   square(side: INTEGER) = RECORD
     number: INTEGER;
     datum: matrix(side,side);
     { Square matrix. }
     END;

VAR
s3: square(3); { Record containing 3X3 matrix. }
i,j: INTEGER;

BEGIN
s3.number : = 3;
FOR i : = 1 TO 3 DO
```

```
    FOR j : = 1 TO 3 DO s3.datum[i,j] : = j;
  FOR i : = 1 TO 3 DO
    FOR j : = 1 TO 3 DO WRITELN( s3.datum[i,j]);
  WRITELN('Size of square(3) in bytes: ', SIZE(square(3))
  );
  END.
```

The output of the program is:

```
 1.00000E + 00
 2.00000E + 00
 3.00000E + 00
 1.00000E + 00
 2.00000E + 00
 3.00000E + 00
 1.00000E + 00
 2.00000E + 00
 3.00000E + 00
Size of square(3) in bytes:       40
```

Array Operations

Except for the special case of PACKED ARRAY OF CHAR (discussed in the preceding section, "String Types"), operations with array types are limited to assignments, either to individual elements or to the entire array, and argument passing.

You can manipulate elements of arrays with *indexed variables*. These are variables that give the name of the array followed by a list of indices in brackets. For example:

```
TYPE
  table2(n: INTEGER) =
    ARRAY[1..n,1..2*n] OF CHAR;
  row(n: INTEGER) =
    ARRAY[1..2*n] OF CHAR;
```

```
VAR
   table,table1: table2(2);
   row1:       row(2);
BEGIN
   .
   .
   table[1][4] : = 'a';   { Put 'a' in row 1, column 4. }
   table[1,4] : = 'a';        { Same thing. }
   .
   .
   table1 : = table;     { Assignment of entire array. }
   row1 : = table[1];  { Assign first row to row1. }
```

Notice that, with a multidimensional array, the indices can be written in consecutive sets of brackets or combined inside one set and separated by commas; the meaning is the same.

The next to last assignment assigns the entire contents of table to table1. The name table, without indices, is a reference to the entire array, and the expression on the right-hand side must also be an entire array with identical type.

The last assignment assigns the contents of the first row of table to row1, since table[1] is an array of four characters.

Internal Representation of Arrays

The elements of an array are stored in "row-major" order. This means that, for example, the two-dimensional array table, type ARRAY[1..2,1..4], is stored as a sequence of elements, and the elements are placed in the sequence with the right-most index varying most rapidly, as shown in Figure 3-15.

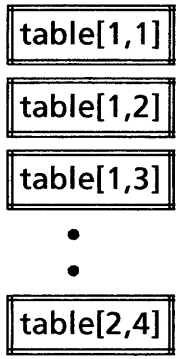


Figure 3-15. Row-Major Order

If the array is PACKED (innermost dimension) and the component type has a boundary requirement of bit alignment, the elements are packed to the bit and the array's boundary requirement is bit alignment. Otherwise, the boundary requirement of the elements is either their natural alignment or byte alignment (whichever is larger), and this is also the boundary requirement of the array itself.

Packed Arrays

Specifying PACKED on an array type has the following consequences:

- If the element type is a suitable ordinal type or small PACKED set, its data representation is changed from the normal one to a packed representation.
- If the element type is as above or is a PACKED record or array with bit alignment, the elements of

the array are packed to the bit, with no fill in between elements.

Except in these cases, PACKED has no effect on data representation. However, it does count in the rules for type equivalence. In the above cases, the general effect is to save storage at the expense of time to access individual elements. A typical use is PACKED ARRAY[1..*n*] OF BOOLEAN, which gives the most compact form of BOOLEAN array.

Note that if a multidimensional array is denoted by

```
PACKED ARRAY[1..2,1..4] OF BOOLEAN;
```

PACKED applies to both dimensions because the element type is BOOLEAN, but in

```
PACKED ARRAY[1..2] OF ARRAY[1..4] OF BOOLEAN;
```

PACKED has no effect on the representation, because the element type is ARRAY without PACKED.

Record Types

A record type represents an aggregate of data, called *fields*, that can have different data types. Records provide a very flexible means of organizing related, but dissimilar information. For example, PACKED records can be used to represent the various bit fields in a hardware device register.

Record Type Definitions

The definition of a record type supplies the types and names of its fields, as shown in Figure 3-16.

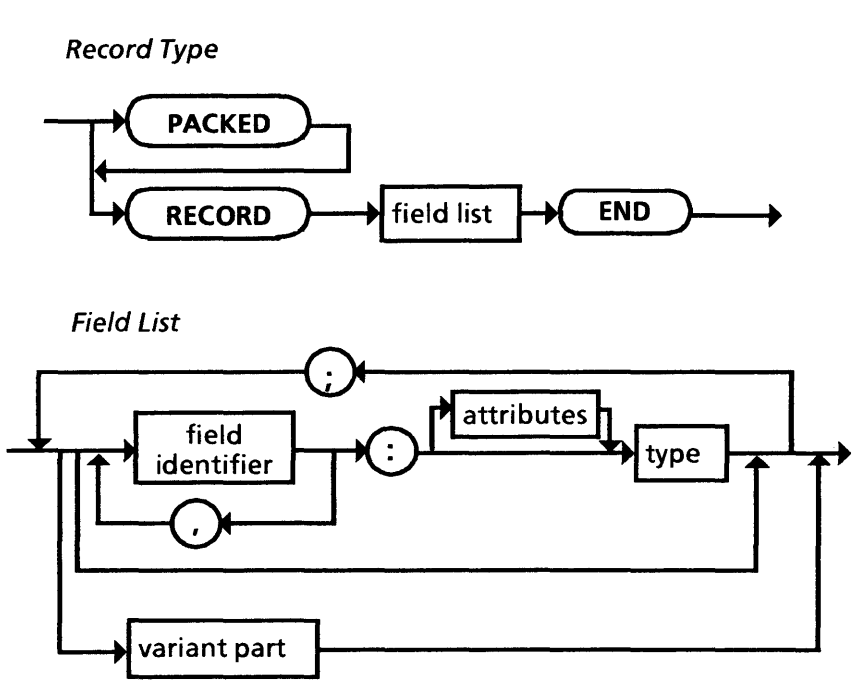


Figure 3-16. Record Type Definition

A record type can be defined as PACKED to specify the most compact storage possible. (The effect of this is discussed later in this section, under "Internal Representation of Records.")

The field list specifies the names and types of the record's fields. Each field identifier is declared as a field name for this record type. The field names must be distinct within the record type, including field names introduced in the variant part. Declaration of an identifier as a field name does not conflict with any other declaration of the identifier, including declaration as a field name in another record type.

The type of a field is specified on the right-hand side of the colon following the field identifier (or list of such identifiers, separated by commas).

Attributes may appear before the type for a field (including a tag field). As explained later in this section, the POS attribute can be used to exactly specify the position of a field in a PACKED record. Note that the POS attribute is not allowed on the tag type of a variant part unless the tag field is actually present. Data representation attributes may be present if allowed on the particular field type. These attributes are defined in the section "Data Representation," later in this chapter.

The variant part of a field list, if present, defines additional fields grouped into variants that share storage in the record. (Variants are described later in this section, under "Records With Variants.")

Operations on Records

An individual field in a record is accessed using field selection or the WITH statement; it is then used in accordance with the field's data type. For example:

```
TYPE employee = RECORD
   name: VARYING_STRING(80);
   salary: REAL;
   END;
VAR
   ref: employee;
```

The fields in the record employee can be accessed by using field selection, as follows:

```
employee.name     { the employee's name }
employee.salary   { the employee's salary}
```

In addition, the WITH statement can be used to establish a reference to the record, such as:

```
WITH ref DO
BEGIN
   name : = 'Anthony Lowell';
   salary : = 25000.00;
END;
```

Note that the only operation applicable to an entire record is assignment of a record value to a record variable.

Records With Variants

A record can contain one or more variants, with each variant containing a group of fields. The variants share storage, so in effect, the record can contain different types of data at different times.

The definition of a record type with variants contains a variant part as the last item in its field list (see Figure 3-16).

The form of the variant part is shown in Figure 3-17.

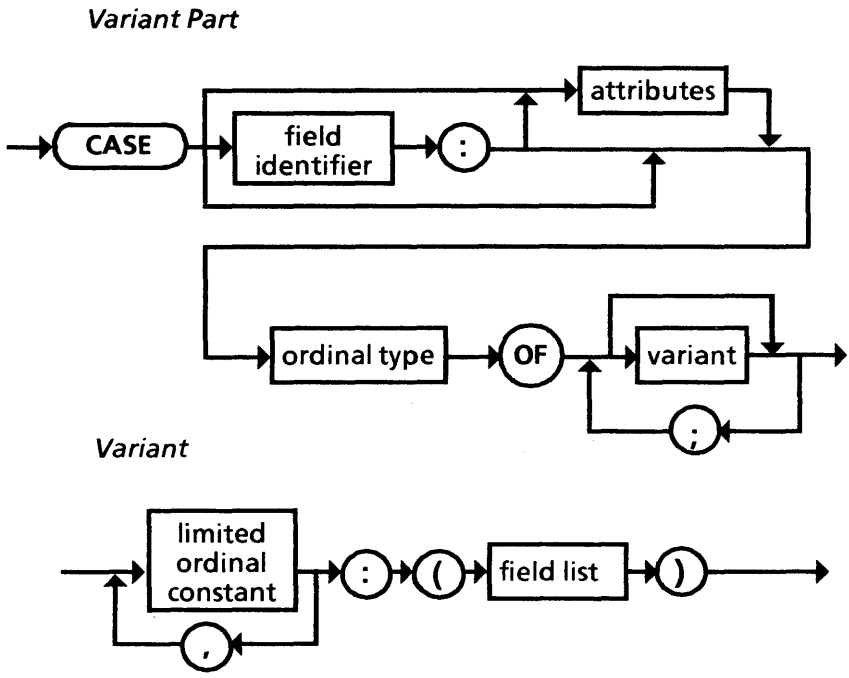


Figure 3-17. Variant Part Syntax

The ordinal type preceding the reserved word OF is the variant part's *tag type*. The POS attribute and data representation attributes may be used with the tag type, as with an ordinary record field, but they have no effect unless the field identifier and colon are present.

When the field identifier and colon are present, the field identifier is declared as a record field in the containing record. (It is not part of any of the variants.) In conventional usage, this explicit *tag field* is set to indicate which variant is currently in use. Note, however, that the compiler never implicitly sets or checks this tag field.

Each variant is preceded by one or more ordina constants (not extent expressions) providing value that may be used to select the particular variant. All o these constants must be compatible with the varian part's tag type, the same value cannot occur twice, an no variant can be preceded by more than 31 constants.

The fields in a particular variant are defined by a fiel list enclosed in parentheses (see Figure 3-16). Note tha this field list may itself contain a variant part therefore, nested variants are possible. Whatever th variant structure, the field names in all variants an the nonvariant part of the record must all be distinc names.

Because the variants in a variant part share the sam storage, assignment to a field in one variant must, i general, be regarded as invalidating the values of al fields in all other variants.

By using knowledge of the record's storage layout (vi the compiler's MAP command qualifier) and takin account of the rules for data representation, you can us fields in different variants at the same time or use then to access the same data by different data types However, this usage is nonstandard and can make program very sensitive to changes in a record type.

Variant records are useful for organizing informatio when the need for some kinds of information depend on other information. For example, the following recor type organizes some medical information, where th requirements differ for males and females:

```
TYPE
   gender: (female,male);
   date = LARGE_INTEGER;
```

```
VAR
  person1:  RECORD
    name: VARYING_STRING(80);
    birthdate: date;
    CASE sex: gender OF
      male:  (); { Empty. }
      female: ( pregnancies: INTEGER)
    END
```

Here, both males and females have birthdates and names, but female variants also have an additional field, containing the number of pregnancies. Since there is a tag field (sex), one or the other variant is identified by assigning either female or male to the field person1.sex.

The field in the female variant is referenced by field selection in the usual way or by using the WITH statement. For example:

```
WITH person1 DO
  pregnancies : = pregnancies + 1;
```

Allocating Records With Selected Variants

Normally, a record occupies enough storage to hold the largest variant in the record's variant part. However, the NEW procedure (see Chapter 9, "VAXELN Routines") can be used to select a specific variant (or a specific set of nested variants) to be allocated. For example:

```
TYPE t = RECORD
  CASE INTEGER OF
    1: (a: array [1..100] OF INTEGER);
    2: CASE INTEGER OF
      21: (b: array [1..100] OF DOUBLE);
      22: (c: array [1..100] OF CHAR))
  END
```

```
VAR p: ^t;

NEW(p,1);       { 400 bytes; holds array a }
NEW(p,2,21);    { 800 bytes; holds array b }
NEW(p,2,22);    { 100 bytes; holds array c }
```

A record created in this way cannot be used in a record assignment; the compiler does not know that it is smaller than the normal size. This or any other reference to a field that does not lie entirely within the allocated part is an unpredictable error.

The SIZE function (see Chapter 9) can be used to determine the amount of storage occupied by a record with selected variants.

Internal Representation of Records

In determining a record's internal representation, the compiler first determines the boundary requirement of each field (including those in variants). This is the maximum of the following:

- The boundary requirement implied by the field's data type.
- Alignment specified by the ALIGNED attribute on a field.
- Byte alignment unless the record is designated PACKED.

The record's boundary requirement is the maximum of all the fields' boundary requirements and any alignment specified by the ALIGNED attribute, if it is present on the record itself. Note the following:

- The boundary requirement and representation of packable data types is different from normal if the record is PACKED.

- The record will have a boundary requirement of bit alignment only if the record is PACKED, all field data types require only bit alignment, and the ALIGNED attribute is not used anywhere in the record.

Once the boundary requirements are determined, the compiler determines a relative location (offset from the beginning of the record) and cumulative size for each field, both quantities starting at zero. Fields are assigned locations in the order of their declarations. "Fill" is introduced between the end of field A and the beginning of field B only to the extent required to ensure that field B has the correct alignment relative to the record's origin. Otherwise, the offset of field B is the same as the cumulative size, including field A.

The cumulative size at the last field is the record's size. Sizes and offsets are kept in units of bits only when they do not equal an integral number of bytes.

The above description applies to records in which all fields have constant size, the record has no variant part, and the POS attribute is not used. The following rules describe how these other cases are handled:

- Variable-size fields are handled by symbolic expressions within the compiler, including expressions to convert units to satisfy boundary requirements.
- If a record has a variant part, the entire variant is treated as the last field in the record. Its boundary requirement is the maximum of all fields in all the variants. Its contribution to the cumulative size is the maximum of the variants.
- A particular variant is laid out starting at the beginning of the variant part.

- If the last field of a record (or any variant) is variable-sized and bit-sized, the record will be variable-bit-sized.
- If the natural size of a record type is ≤ 32 bits, its size may be specified exactly by using the BIT, BYTE, WORD, or LONG attribute. BYTE, WORD, and LONG force at least byte alignment.
- The POS attribute (see below) forces a field to have a particular constant offset (expressed in bits within the attribute). The compiler issues an error message if the size conflicts with the field's boundary requirement or the cumulative size, including the preceding field.

Note: The actual layout of a record can be determined with the compiler's MAP command qualifier.

POS Attribute

The POS attribute can be used to specify that a field within a PACKED record begins at a given position with respect to the origin. It conflicts with explicit use of the ALIGNED attribute.

The syntax for the POS attribute is shown in Figure 3-18.

Figure 3-18. POS Attribute Syntax

The extent expression in the syntax for the POS attribute must produce a nonnegative integer constant.

The POS attribute can be applied to any field within a PACKED record. The field with attribute POS(*n*) begins at bit position *n*; the origin of the record is POS(0). The position *n* must be beyond the last bit of the previous field, if any. The bit offset must be consistent with the alignment requirements of the field. For instance, [POS(15)] means that the field begins at bit 15 (the 16th bit).

Generally, the POS attribute introduces filler bits or bytes in the record. When used within a variant, the attribute denotes the position relative to the beginning of the entire record.

When applied to a type governing more than one identifier, POS applies to the first. For example:

```
a,b : POS(32) CHAR;
```

Here, the field a is 32 bits from the record's origin; b is 40 bits from the origin.

Pointer Types

A pointer value is either the address of a data item in the job's virtual memory or a distinct null value denoted by the reserved word NIL. In Pascal, a pointer type definition normally specifies the type of the items to which the pointers will point (although ↑ ANYTYPE is also allowed). This is referred to as the pointer's *associated data type.*

When a pointer variable has a valid value, you can make a reference to the located variable by suffixing the pointer variable with the indirection operator (@, ^, or ↑). For example:

```
TYPE stringptr = ↑STRING(80);
VAR p : stringptr;
BEGIN
```

```
    NEW(p);
    .
    .
    p↑ := 'any string';
END;
```

Here, p is a pointer to a variable of type STRING, and p↑ denotes that variable. The NEW procedure allocates storage large enough to hold an 80-character string and assigns its address to p. The assignment statement then assigns 'any string' to the data item referenced by p↑.

Pointer values other than NIL are obtained from the routines NEW and ADDRESS and from some of the kernel procedures such as CREATE_MESSAGE. A pointer value so obtained remains valid only as long as the storage it addresses remains allocated in the job. Thus, a pointer obtained from NEW becomes *invalid* when the storage is deallocated by a call to DISPOSE. Pointers obtained from the ADDRESS function are subject to further restrictions, as discussed in the description of that function.

Pointer values can be compared with the equality and inequality operators, with each other, or with NIL. Pointer variables are assignment compatible only with NIL and with other variables of the same type or type ↑ANYTYPE (which is explained later in this section).

Pointer Type Definitions

The definition of a pointer type supplies the data type of the variables whose locations it can address. A pointer type definition can be used in the following contexts:

- On the right-hand side of a TYPE declaration.
- As the type of a parameter.

- As the type of a function result.
- As the target type of a typecast variable.

The syntax for a pointer type definition is shown in Figure 3-19.

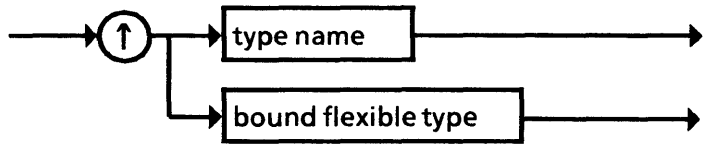


Figure 3-19. Pointer Type Definition

The type name in a pointer type definition must be the name of a nonflexible data type. It specifies the type of the items to which the pointers will point. The type ANYTYPE can be used here to declare pointers of unspecified type, as explained later in this section.

A pointer type definition can also specify that pointers of the defined type will point to items of a bound flexible type. Bound flexible types are discussed under "Flexible Types," earlier in this chapter.

The following example shows the use of a pointer type to identify dynamically allocated records:

```
PROGRAM update(INPUT, OUTPUT, PFILE);
  TYPE
    persrec = RECORD
      name: VARYING_STRING(80);
      salary: REAL;
    END;
```

```
VAR
  perspointer: ^persrec;
  pfile: FILE OF persrec;
BEGIN
  NEW(perspointer);
    { Allocate new record; its location is now in
    perspointer. }
  WRITE('Enter name, salary: ');
  READLN(perspointer^.name,
  perspointer^.salary);
  .
  .
  REWRITE(pfile);
  WRITE(pfile,perspointer^);
  DISPOSE(perspointer);
    { Dispose of the record; perspointer is now
    invalid. }
END.
```

Internal Representation of Pointers

Pointer variables are longwords containing 32-bit addresses. NIL is represented by zero, which is never a valid memory address in a process.

ANYTYPE Data Type

The data type ANYTYPE represents data of completely unspecified type (unlike BYTE_DATA, where the data's size is specified). ANYTYPE may only be used for VAR parameters and for the associated data type of a pointer type.

When a ↑ ANYTYPE pointer is used to reference data, the data item's type must be specified by *typecasting*, as explained in Chapter 5, "Variables."

For example,

```
p↑::INTEGER
```

casts the pointer p to type INTEGER when it is used.

Note that ↑ANYTYPE is assignment compatible (as either the source or target of the assignment) with any other type of pointer.

File Types

In Pascal, a file type is the type of a Pascal *file variable*, which is a data item used to designate the source (target) of an input (output) operation. In many cases, the I/O source or target will be a true file (that is, a file in the file system), and it may be convenient to ignore the distinction between the Pascal file variable and the file system file. However, they are not at all the same thing, and file variables may be used independently of the file system.

A file variable has an associated data item called a *file buffer*, which holds the data transmitted by use of the GET or PUT procedures. The file buffer's data type is called the file's *component type*. In I/O involving a true file, this is the data type of the records in the file. The component type VARYING_STRING(n) may be used to handle files with variable record lengths.

The predeclared file type TEXT has special properties useful in text I/O. Textfiles are explained in Chapter 15, "Input and Output."

The remainder of this section discusses the definition of file types, restrictions on file variables, and the internal representation of file variables. The use of file variables in I/O is fully described in Chapter 15.

File Type Definitions

A file type definition has the form shown in Figure 3-20.

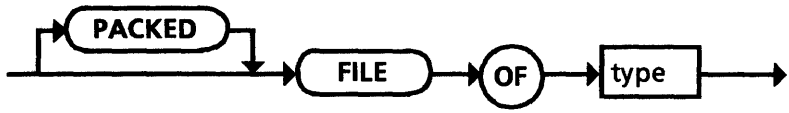


Figure 3-20. File Type Definition

The type in a file type definition determines the file's component type, and hence the type of its associated buffer. It must be constant-sized, with the size not exceeding 32,768 bytes. The type must not be a file type (that is, there cannot be "files of files") or a record or array type with components of a file type (there cannot be "files of arrays of files," and so forth).

PACKED has no effect in VAXELN Pascal, except in regard to the equivalence of file types, as explained later in this chapter.

The following examples illustrate file type definitions:

```
TYPE
   pers_file = FILE OF RECORD
       name: VARYING_STRING(80);
       hire_date: LARGE_INTEGER;
       salary: REAL;
       END; { A data type used for files of employee
       records.}
VAR
   console_in: FILE OF CHAR;{ File of characters. }
```

Restrictions on File Variables

A variable of a file type does not have a value in the ordinary sense. It is a control block used by the system to store information for supporting the various file I/O operations. For this reason, file variables are not allowed as targets of assignment statements or as value parameters.

Any modification of the storage of a file variable, except with the predeclared I/O operations, has unpredictable effects. For this reason, file variables should usually not be used within the variant parts of variant records. (The compiler issues a warning message in this case.)

The contents of the buffer variable are closely related to the operations on the file variable. For this reason, it is an error to refer to the buffer variable and perform any operation on the buffer variable in the same simple statement. For example, if p is a procedure with VAR parameters of file types, then p(file, file↑) is generally invalid. Again, violations of this rule have unpredictable effects.

Internal Representation of File Data

A file variable is byte-aligned and occupies 16 bytes plus the size of its buffer variable's type (rounded up to the nearest byte if it is bit-sized). A file variable's address is used by the run-time library to uniquely identify a file. Note that file variables are shareable by a job's processes only if they are declared at the outer level.

The structure of the internal representation of a file variable is shown in Figure 3-21.

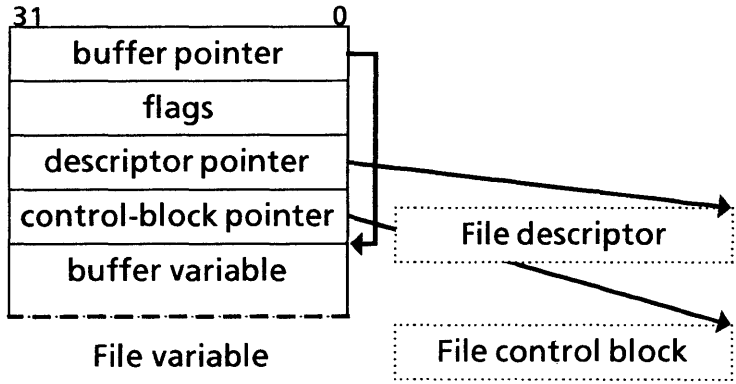


Figure 3-21. Internal Representation of a File Variable

The flags part of the file variable contains such information as the file's location in the PROGRAM argument list (if applicable), whether the buffer is currently "valid" in Pascal terms, and so forth.

The file descriptor contains information about the file that is determined by the compiler, such as whether it has type TEXT, whether it is declared in a module's outer block, and so forth. Usually, the file descriptor is allocated in read-only storage.

The file control block is allocated dynamically by the run-time routines. It contains internal information needed by the routines to process file I/O, such as the size and address of the record buffer, whether the file is the standard INPUT or OUTPUT, whether access is sequential or direct, and the PORT value identifying the circuit used for data transmission.

The exact definitions of the file variable's parts, the file descriptor, and the file control block are in the source file PASIODEF.PAS.

System Data Types

The system data types PROCESS, AREA, EVENT, SEMAPHORE, MESSAGE, PORT, NAME, and DEVICE are used to synchronize the concurrent parts of programs (*processes*), to control devices, and to communicate between programs.

Each system data type represents the identifying value of a VAXELN kernel object. A VAXELN Pascal program declares variables of these types to hold the identifying values of the corresponding objects.

Values of the system data types can be assigned to variables of the same type, passed as arguments to parameters of the same type, and returned by functions with the same result type. Generally speaking, all other operations on these types are performed by predeclared kernel procedures.

The system data types are briefly described below, including the internal representation of each type. The kernel objects themselves (including their internal representation) are described in detail in the *VAXELN User's Guide*.

PROCESS Data Type

A PROCESS object represents an independent thread of execution; that is, a process. There can be any number of processes executing the same process block.

PROCESS values are represented internally as 32-bit longwords that are used by the kernel to identify an individual thread of execution. They are valid only within their own job.

AREA Data Type

An AREA object represents a region of memory that can be shared among jobs on a single node in a VAXELN network.

AREA values are represented internally as 32-bit longwords that are used by the kernel to identify a particular area and its associated properties.

EVENT Data Type

An EVENT object represents the state of an event used for process synchronization.

EVENT values are represented internally as 32-bit longwords that are used by the kernel to locate the actual data and its associated properties.

SEMAPHORE Data Type

A SEMAPHORE object represents a synchronization gate used to meter process execution and synchronize access to shared data.

SEMAPHORE values are represented internally as 32-bit longwords that are used by the kernel to locate the actual object and its associated properties.

MESSAGE Data Type

A MESSAGE object describes data transmitted between processes. Messages can be transmitted between processes on the same network node or on different network nodes.

MESSAGE values are represented internally as 32-bit longwords that are used by the kernel to identify a particular message and its associated properties.

PORT Data Type

A PORT object represents a repository for messages waiting to be received; that is, a *message port*. Only the processes in the job that created a port can receive a message from that port; any process in any job can send a message to it.

PORT values are represented internally as 128-bit quantities that are used by the kernel to uniquely identify the message port.

NAME Data Type

A NAME object represents a user-defined name for a message port.

NAME values are represented internally as 32-bit longwords that are used by the kernel to identify a particular name for a port.

DEVICE Data Type

A DEVICE object represents a device interrupt connected to an interrupt service procedure.

DEVICE values are represented internally as 32-bit longwords that are used by the kernel to locate the actual object containing its associated properties.

Other Predeclared Data Types

The data types described in this section do not fall into any of the preceding categories of predeclared data types. These types are BYTE_DATA and LARGE_INTEGER.

BYTE_DATA Data Type

The type BYTE_DATA(*n*) represents storage of a specific size (*n* eight-bit bytes), whose contents are not interpreted in any specific way. There are no operations defined on this type except assignments and argument passing.

To facilitate systems programming, some special conventions apply to byte data:

- Routine parameters of type BYTE_DATA(*n*) are considered to be compatible with any data type of the same size.
- Conformant BYTE_DATA parameters, such as BYTE_DATA(<*n*>), are compatible with data of any size.

The size parameter of BYTE_DATA can be omitted in a typecast variable, and the result is interpreted as BYTE_DATA, with the size of the source variable. For example, if variable veloc is of type REAL, the variable

```
veloc::BYTE_DATA
```

is interpreted as BYTE_DATA(4), because REAL data requires four bytes of storage.

LARGE_INTEGER Data Type

The data type LARGE_INTEGER represents signed, 64-bit integers. Variables of this type are declared with the word LARGE_INTEGER. The data type is provided primarily for representing 64-bit time values.

Only a few operations are allowed on LARGE_INTEGER values, and they are not ordinals. (See Chapter 6, "Expressions and Operators," for the operations allowed.)

The range of LARGE_INTEGER data is from -2^{63} to $2^{63}-1$ (approximately $\pm 9.2233 \times 10^{19}$). Any attempt to compute a LARGE_INTEGER value outside this range causes an integer overflow exception, with the result undefined.

Internal Representation of LARGE_INTEGER Data

Internally, a LARGE_INTEGER data item is represented in 2's complement form in a VAX quadword (64 bits). The most significant bit (63) is always zero for positive numbers and always one for negative numbers.

The internal representation is shown in Figure 3-22.

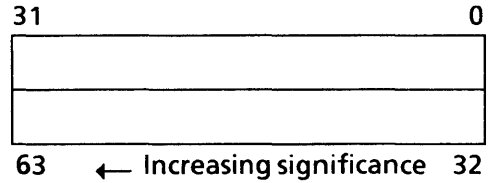


Figure 3-22. LARGE_INTEGER Representation

Type Equivalence

VAXELN Pascal imposes the requirement of *type equivalence* in several contexts, the most important being passing an argument to a VAR parameter, assignment of array and record values, and assignment of pointer values where the associated ("pointed-to") types must be equivalent.

The general principle underlying type equivalence is that two data types should be considered equivalent if they have the same internal representation and they

have the same logical meaning in Pascal terms. The interpretation of this principle for the various Pascal data types reflects practical usage of the language, but it may not exactly match the requirements of a particular application. In cases where the type equivalence rules are too strict, typecasting may be used to override them.

Type equivalence is a weaker relation than *type identity*. Two types are identical if they are the same predeclared type or denote the same occurrence of one of the syntactic categories whose name ends in "type definition." For example:

```
TYPE A = 1..3;
     B = 1..3;
     C = A;
```

Here, types A and C are the same, but B is a distinct subrange type. This technical notion of identical type is needed only for the definition of type equivalence.

Type equivalence is a stronger relation than *assignment compatibility*, which is summarized in Chapter 7, "Pascal Statements." For example:

```
VAR  A : REAL;
     B : DOUBLE;

  A := B;
  B := A;
```

Here, the assignment statements are valid because REAL and DOUBLE are compatible types, even though they are not equivalent.

The type equivalence rules for arrays and flexible types involve extent values. Failure of type equivalence related to extent values is a range violation. If the extent values are not constant, the error will only be detected at run time, and only if range checking is

enabled. In all other cases, type equivalence errors are detected by the compiler.

The following subsections present the equivalence rules for the various classes of data types.

Ordinal Types

The *base ordinal types* are INTEGER, CHAR, BOOLEAN, and enumerated types. In the most general terms, an ordinal type has a base ordinal type, may be a subrange type (for example, "0..7") and may have a representation attribute, such as BYTE. Every ordinal type has a minimum value and a maximum value.

Two ordinal types are equivalent if all the following are true:

- They have the same base ordinal type.
- Either both are subranges or neither is.
- They have the same representation attribute, if any.
- They have the same minimum and maximum values.

If the two types have the same base type, lack of equivalence is only a warning-level error in some cases where the internal representation is the same. For example:

- Neither data item is PACKED.
- One data item is PACKED, but is specified with BIT, BYTE, WORD, or LONG, where that is the normal representation of the base type.
- Both data items are PACKED and have the same representation attribute, if any.

A program giving this sort of warning message may be correct in practical terms, but you should be sure that

the actual values of variables do not violate the variables' ranges. Range-checking code is not generated in these cases.

Set Types

Two set types are equivalent if all the following are true:

- They have the same element type.
- Either both are PACKED or neither is PACKED.
- Both have the same representation attribute, if any.
- They have the same minimal element and the same maximal element.

If two set types have the same base type, lack of equivalence is only a warning-level error in some cases where the internal representation is the same. For example:

- Neither set is PACKED and both occupy the same number of longwords.
- One set is PACKED but is specified with LONG, and the other occupies one longword.
- Both sets are PACKED and are specified with the same representation attribute, if any.

Flexible Types

Each definition of a flexible type introduces a distinct type that is not identical with any other. For example:

```
TYPE
   A (m: INTEGER) = ARRAY [1..m] OF INTEGER;
   B (m: INTEGER) = ARRAY [1..m] OF INTEGER;
```

Here, types A and B are *different* types.

A bound flexible type is a flexible type with explicit extent values. For example:

```
TYPE
   T1 (m: INTEGER) = ARRAY [1..m] OF INTEGER;
   T2 = T1(10); { T1(10) is a bound type. }
```

Here, the definition of type T2 is a bound flexible type.

If two bound types are matched for equivalence, and neither is defined in terms of the other, the only thing that matters is whether the types "within" them are equivalent.

For example:

```
TYPE
   T1 (m: INTEGER) = ARRAY [1..m] OF INTEGER;
   T2 (n: INTEGER) = ARRAY [0..n] OF INTEGER;
   T3 = T1(10); { T1(10) is a bound type. }
   T4 = T2(9); { T2(9) is a bound type. }
```

Here, T3 and T4 are equivalent because the two arrays, although they are not identical, have the same number of INTEGER elements.

If one of the types is defined in terms of the other, then equivalence is determined by comparing extent values in corresponding positions. For example:

```
TYPE
   T1 (m,n: INTEGER) = ARRAY [1..m,1..n] OF
   INTEGER;
   T2 (o,p: INTEGER) = T1(o,p);
   T3 = T1(5,10);
   T4 = T2(5,10);
```

Here, T4 = T2(5,10) is "expanded" to the type T1(5,10) by substituting the extent values for o and p. Here, T3 and T4 are equivalent. Nonequivalence of flexible types due to differing extent values is a range violation.

Predeclared Flexible Types

For the data types STRING, VARYING_STRING, and BYTE_DATA, two types are equivalent only if the type is the same and the extent value is the same. For example, STRING(10) is equivalent only with STRING(10). It is a range violation if the extents are different.

Predeclared Non-Flexible Types

If a predeclared type is defined in this manual as a particular array or pointer type, then it is covered by the regular rules for the equivalence of array and pointer types, which follow. Apart from this, two predeclared non-flexible types are equivalent only if they are identical.

Array Types

Array types are equivalent if the following requirements are satisfied:

- They have the same number of dimensions.
- They have equivalent element types.
- They have the same alignment requirement. (This only matters if the ALIGNED attribute is used.)
- Both are PACKED or neither is PACKED in corresponding places.
- The number of elements is the same for each dimension. It is a range violation if the numbers of elements are different.

Note: The last requirement is satisfied if the bounds are the same in each dimension, but that is not required for equivalence. For example, ARRAY ['a'..'z'] OF INTEGER is equivalent to ARRAY[1..26] OF INTEGER, since the number of elements is 26 in either case.

Record Types

Two record types are equivalent only if they are identical.

Each definition of a record type introduces a distinct type that is not identical with any other. For example:

```
TYPE
  A = RECORD
    name : STRING(80);
    salary : REAL;
  END;
  B = RECORD
    name : STRING(80);
    salary : REAL;
  END;
  C = B;
```

Here, A and B are *different* types; C is the same as B because it is only a synonym. Note again that A and B are *not* identical even though they have the same field definitions.

Pointer Types

If one or both types are ↑ANYTYPE, they are equivalent; otherwise, the associated types must be equivalent.

File Types

Two file types (other than TEXT) are equivalent if the following requirements are satisfied:

- Both are PACKED or neither is PACKED.
- They have equivalent buffer-variable types.

The file type TEXT is not equivalent to any other type.

Data Representation

For some system programming applications you may need to know the rules by which data is stored internally, and the language features, such as data attributes, that modify the representation.

Each data type has an internal representation that is used for entire variables of that type. It specifies the following:

- The size of the data item in bits or bytes.
- The boundary requirement of the type (bit alignment, byte alignment, and so forth).
- The actual form of a data item in memory.

The normal internal representation for each type is given in the section on that type earlier in this chapter. The representation can be modified as follows:

- Within a PACKED record or array, the following packable data types have a special representation (unless overridden by one of the data size attributes, as explained later in this section): BOOLEAN, enumerated types, subranges of INTEGER and CHAR, and small PACKED sets. The special packed representation is explained in the section for each type.
- Certain ordinal data items, small PACKED sets, and PACKED records may have their size modified by one of the data size attributes BIT, BYTE, WORD, or LONG, as explained later in this section.
- The ALIGNED attribute may be used to specify a more stringent than normal alignment requirement for arrays, records, and fields within records.

- In the VAX argument list generated for a routine call, the argument corresponding to a value parameter of type CHAR, BOOLEAN, or enumerated is a longword whose low-order byte or word contains the value (with the normal representation) and whose high-order bytes are zero. This means that the entire longword can be used as the integer value ORD(argument).
- The compiler may keep intermediate expression results in other forms, which are not visible to you.

The remainder of this section discusses the following topics:

- The definition of *boundary requirement*
- The definition of *size*
- Packed data
- The data size attributes BIT, BYTE, WORD, and LONG
- The ALIGNED attribute

Boundary Requirement

The *boundary requirement* of a data type is the *minimum* alignment for storage allocation of that type. For example, if the boundary requirement for a data type is byte alignment, data of that type must be allocated on byte boundaries. The compiler may allocate the data item on "higher" boundaries, such as longword boundaries for integers.

If a data type's boundary requirement is bit alignment, it can be allocated at any location.

Size of Data

The *size* of a data type is the amount of storage required to represent a data item of that type. The size is normally an integral number of eight-bit bytes, and those are the units returned by the SIZE function. Bit-sized data is possible in the following cases:

- Within a PACKED record or array, data types with special packed representations, as mentioned previously, may be bit-sized.
- A PACKED record or array containing bit-sized data may be bit-sized.
- Data items with the BIT attribute may be bit-sized.

Note: The storage allocated for a bit-sized record or array data item is rounded up to the next integral number of bytes, unless the item is itself immediately contained in a PACKED record or array. The extra, or "fill," bits resulting from this allocation are not properly part of the data item and have unpredictable values.

Packed Data

The reserved word PACKED can be used to control data representation in the following ways:

- Applied to a record type denotation, PACKED causes fields of packable data types to be stored in their special packed representation. To the extent that the fields have a boundary requirement of bit alignment, they are packed so that there are no unused bits.
- If PACKED is specified in a record type definition, the BIT, BYTE, WORD, or LONG attribute can

also be used on the record definition, and POS may be used within the record.

- If the element type of an array definition is one of the packable data types and the array is designated PACKED, each element is stored in its special packed representation. If the element's data type has a boundary requirement of bit alignment, the array is packed so that there are no unused bits between elements.
- If a set type definition is designated PACKED, and if the type's normal representation is one longword (that is, the ordinal number of the maximum element is less than 32), then the set type is a packable data type.

Apart from these cases, PACKED has no effect on the data representation. It *is*, however, significant in the rules for type equivalence whether or not the representation is changed. In addition, PACKED ARRAY[1..*n*] OF CHAR can be used as a string, as discussed earlier in this chapter.

Data Size Attributes

The BIT, BYTE, WORD, and LONG attributes can be used to control the size of an ordinal data item, a small set item, or a small PACKED record. In terms of VAXELN Pascal syntax rules, these attributes are allowed preceding a type on the right hand side of a type declaration, in a VAR declaration, in a field of a record (including the tag following CASE in the variant part), or preceding the type definition of an array element.

Generally, these attributes are intended to be used for fields in data structures. Indiscriminate use of them can result in substantial performance degradation.

The type to which the data size attribute is applied must be one of the following:

- A PACKED record type definition whose natural size is constant and ≤ 32 bits.
- A set type definition (PACKED or not) defining a *small* set type, where small means the ordinal number of the maximum element is less than 32.
- An explicit subrange ("low-value..high-value").
- A named ordinal type or named small set type. In this case, if the named type already has a data size attribute, the new attribute overrides it.

Additionally, the following properties and rules apply to the data size attributes:

- A named type defined using one of these attributes may be used anywhere, subject to the normal data type rules. In particular, such a named type may be used in typecasting. Typecasting of bit-aligned and packed fields is allowed, providing the cast type is appropriate. (See Chapter 5, "Variables," for a discussion of typecasting.)
- When a named type has one of these attributes, any item defined directly as of the named type will also have the attribute.
- If an item of type CHAR has one of these attributes, it is not accepted as a string.
- If an enumerated type is originally defined with a representation attribute, subranges of the type will have the same attribute, unless overridden.
- If you typecast a reference to a data type with the BIT attribute, or to an array or record type with an alignment requirement of only bits, the reference

doesn't have to be addressable; that is, it may appear to the compiler to have a bit offset.

- If the control variable of a FOR loop is INTEGER or BOOLEAN, it must not have a representation attribute other than LONG (INTEGER) or BYTE (BOOLEAN). If the control variable is CHAR or enumerated, it can have BYTE, WORD, or LONG, but not BIT.
- If a field of a PACKED record or the element type of a PACKED array has one of these attributes, its representation is not affected by the PACKED designation.

BIT Attribute

The BIT attribute specifies the number of bits of storage occupied by a data item, and it implies a boundary requirement of bit alignment. It conflicts with the BYTE, WORD, LONG, and VALUE attributes.

The syntax for the BIT attribute is shown in Figure 3-23.

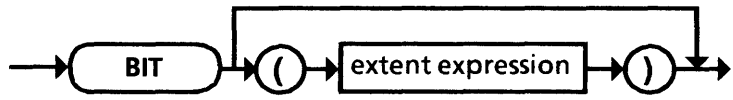


Figure 3-23. BIT Attribute Syntax

The extent expression in the syntax for the BIT attribute must produce an integer constant in the range 1..32. The constant must be at least as large as the data item's natural size in packed representation.

If the bitfield's size exceeds the data item's required size, the field value is extended as follows:

- Signed integer fields are sign extended.
- Other ordinal fields are zero extended.
- Sets are zero extended.
- For PACKED records, the extra bits have undefined values unless a defined value, such as ZERO, is assigned to the entire record.

The BIT attribute makes it possible to precisely describe any packed representation while keeping subranges and set ranges in line with the values actually expected.

BYTE Attribute

The BYTE attribute specifies that a data item occupies exactly one byte of storage, and it implies a boundary requirement of byte alignment. It conflicts with the BIT, WORD, and LONG attributes.

Note that the data item's size without BYTE must not exceed eight bits.

WORD Attribute

The WORD attribute specifies that a data item occupies exactly one word of storage, and it implies a boundary requirement of byte alignment. It conflicts with the BIT, BYTE, and LONG attributes.

Note that the data item's size without WORD must not exceed 16 bits.

LONG Attribute

The LONG attribute specifies that a data item occupies exactly one longword (32 bits) of storage, and it implies a boundary requirement of byte alignment. It conflicts with the BIT, BYTE, and WORD attributes.

Note that the data item' size without LONG must not exceed 32 bits.

The ALIGNED Attribute

The ALIGNED attribute may be specified on an array type definition, a record type definition, or on the type of a field in a record definition. Note that in the latter case, the POS attribute cannot be used.

The syntax for the ALIGNED attribute is shown in Figure 3-24.

Figure 3-24. ALIGNED Attribute Syntax

The extent expression in the syntax for the ALIGNED attribute must produce the integer constant 0 (byte alignment), 1 (word alignment), or 2 (longword alignment).

The net boundary requirement of the array, record, or field is the maximum of the data item's natural boundary requirement and whatever is specified by the ALIGNED attribute. Only the net boundary requirement figures in the rules for type equivalence.

Chapter 4
Constants

Introduction

VAXELN Pascal provides a variety of means for describing constant data; that is, data whose value does not change during the execution of a program. This chapter presents the rules for literal constants, the declaration of named constants, and initializers.

A *literal constant* is a lexical token denoting a particular ordinal, floating-point, or character string value. A *named constant* is a name (identifier) denoting one of these same types of values. Named constants are declared in CONST declarations and in enumerated type definitions. In addition, VAXELN Pascal provides several predeclared named constants, which are summarized in the last section of this chapter.

In the declaration of named constants and in most other places where standard Pascal requires a constant value, an extent expression with constant operands can be used to supply an ordinal value. This feature is explained under "Flexible Types" in Chapter 3. The few cases where the language requires a constant rather than a general restricted expression are explained in the section "Limited Ordinal Constants," later in this chapter.

An *initializer* is a special syntactic construction used to specify a constant initial value for a variable (or default value for a value parameter). Initializers can be used with any data type, including record and array types.

Certain forms of constant data are defined using variable declarations, whose syntax is given in the next chapter. The most important case is a VAR declaration with the READONLY attribute. Such a declaration allocates readonly storage for the declared data, which can be of any type. The data's value is given by an initializer.

A VAR declaration with the VALUE attribute may be used (for certain data types) to declare a constant value made available to the VAX/VMS linker. Used with the EXTERNAL attribute, it allows access to such items defined in other languages; for example, assembly language. Note that such items are not full-fledged constants, since their values are not known at compile time.

Literal Constants

Literal constants are lexical tokens that denote a particular value of type INTEGER, CHAR, REAL, DOUBLE, or STRING. The type is implied by the form in which the literal constant is written and, to some degree, the context in which it is used. Numbers are unsigned simply because the sign is a distinct lexical token.

Literal Integer Constants

Literal integer constants can be written either in decimal or nondecimal form; the default is decimal. In most places, the syntax allows an integer-valued constant expression in lieu of an integer constant.

Decimal Literals

An unsigned decimal integer literal consists of 1 to 31 decimal digits specifying a nonnegative value in the range 0 through $2^{31}-1$. The following are valid examples:

```
102938747
398
0
1
12
```

The type of such a literal constant is INTEGER.

Note that 2,147,483,648 is not a valid integer literal, even though −2,147,483,648 is (internally) an acceptable value. This most negative integer value can be specified by the hexadecimal literal %X80000000.

Also note that any literal containing a decimal point (.) is of type REAL, not of type INTEGER.

Nondecimal Literals

Integer literals can also be written in binary, octal, or hexadecimal form. Each of these forms consists of a radix specifier followed by an unsigned series of digits which are optionally enclosed in apostrophes.

The radix specifiers are %B (binary), %O (octal), and %X (hexadecimal). It makes no difference whether the radix specifiers are uppercase or lowercase. In the quoted version, spaces and tabs can be used for clarity and are ignored. Spaces and tabs can also appear between the radix specifier and number.

This form of literal specifies an integer by giving its two's-complement representation with zero extension to 32 bits. It thus denotes a value *n* in the range

$-2^{31} \leq n < 2^{31}$. The following are valid examples, with their decimal equivalents shown in {}:

```
%B'10001101'      { -115 }
%B001101          { -51 }
%O'1777377'       { -1573121 }
%O71263           { -3405 }
%X'981FA'         { -425478 }
%xFF012           { -4078 }
%XFFFFFFFF        { -1 }
%x80000000        { -2147483648 }
```

Literal CHAR Constants

A single character enclosed in apostrophes (for example, 'a') is a literal constant of type CHAR.

Literal Floating-Point Constants

Figure 4-1 uses the same notation as the language syntax diagrams to show the form of a literal floating-point constant. Here, the terminal elements are decimal digits, the period used as a decimal point (.), the plus (+) and minus (-) symbols, and the lowercase letter e or the uppercase letter E.

A literal floating-point constant has a *mantissa,* consisting of an integral part and a fractional part (separated by a decimal point), and an optional *exponent.* Note that the integral part is required, and that the constant must contain a fractional part or an exponent. The total number of integral and fractional digits must not exceed 34, and the total number of characters in a literal must not exceed 43.

The exponent consists of the letter E (or e) followed by an optionally signed integer exponent. The fractional part of the mantissa is optional only if the exponent is specified.

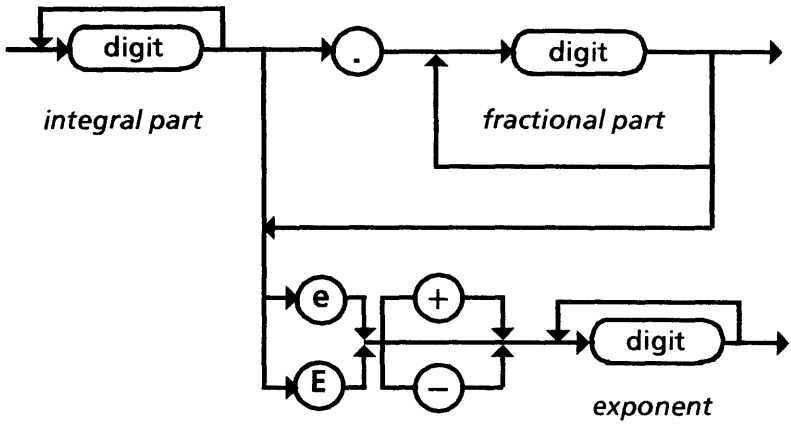


Figure 4-1. Literal Floating-Point Constant Syntax

The exact decimal value of a floating-point literal is equal to the mantissa multiplied by the power of 10 indicated by the exponent. For example, the literal floating-point constant 1.234E2 is equivalent to the expression 1.234×10^2, which is the decimal number 123.4. The exact value is converted by rounding to the closest value in the internal F, D, or G_floating format, depending on the context.

For purposes of determining the result type of a floating-point operation, a floating-point literal is considered to have the type REAL. However, constants are not converted to an internal representation until the compiler determines the context of use. This applies to named floating-point constants as well as to literals. Thus, no accuracy is lost if a floating-point constant with many digits is used in a DOUBLE expression or assigned to a DOUBLE variable.

The following are valid examples of literal floating-point constants:

```
1.0
1E10
1e - 10
1E + 10
1.234
0.234e10
1.234E - 10
1.234e + 10
0.234
```

The following constants are invalid:

```
1.    { Decimal point without fraction; should be 1.0}
.234 { No integral part; should be 0.234 }
0.12345678901234567890123456789012312441 2e10
      { Too long. }
```

Literal String Constants

A *string constant* is a string enclosed in apostrophes. For example:

```
'This isn''t a character, it''s a string.'
```

Notice that the apostrophe itself is represented as a double apostrophe in a string constant. The string constant, including the surrounding apostrophes, must be written on one line in your programs.

A string of length zero is called the *null string* and is represented by an empty string constant:

```
''
```

Note that this is different from the literal CHAR constant '□', where □ represents a space. Note also that the null string is not the same as the character NUL (CHR(0)).

The type of a string constant is CHAR if it contains one character; such constants can be assigned to CHAR variables and used in contexts where ordinal types are required; for instance, as arguments to the ORD function. It is treated as a string when appropriate; for example, when it is assigned to a STRING variable.

Nonprinting Characters in Constants

Nonprinting characters can be represented by their ordinal values (see Table 3-1 in Chapter 3, "Data Types") enclosed in parentheses, concatenated to string constants:

```
'These are printable.'(7)'<-that isn''t!'
```

Any ordinary string constant can be followed by a list of decimal integer literals enclosed in parentheses. Each literal is the ordinal value of a character, and at least one must be present. If more than one literal is present, they are separated by commas.

Although the ordinary constant must be written on one line, these parenthesized lists can be separated by "white space" (including line breaks) and can contain white space. For example, all of the following constants are valid:

```
'printable'(7,7,7)

'printable'
(7,7,7)'printable'

'printable'
(7 ,7,7
)'printable'
```

Each of these is a single string constant.

Constant Declarations

Figures 4-2 and 4-3 show the syntax for constant declarations. In Figure 4-2, the constant identifier is declared as a constant name whose value is given by the constant on the right-hand side of the equal sign.

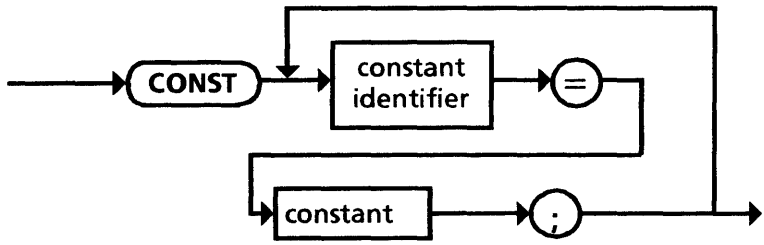


Figure 4-2. Constant Declaration Syntax

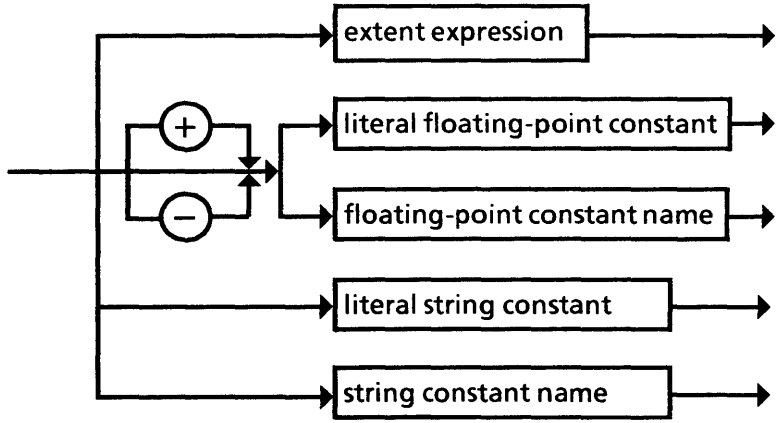


Figure 4-3. Constant Syntax

In Figure 4-3, the extent expression must have constant operands. It can be a literal or named ordinal constant, a signed integer constant, or a more general expression. Extent expressions are explained under "Flexible Types" in Chapter 3.

Limited Ordinal Constants

There are three contexts in the languages's syntax where an ordinal constant value must be specified using the limited ordinal constant syntax shown in Figure 4-4, rather than the more general extent expression. These contexts are:

- Specifying the limits in a subrange type.
- Specifying the ordinal value associated with a variant in a record type definition.
- Specifying the case value of a statement within a CASE statement.

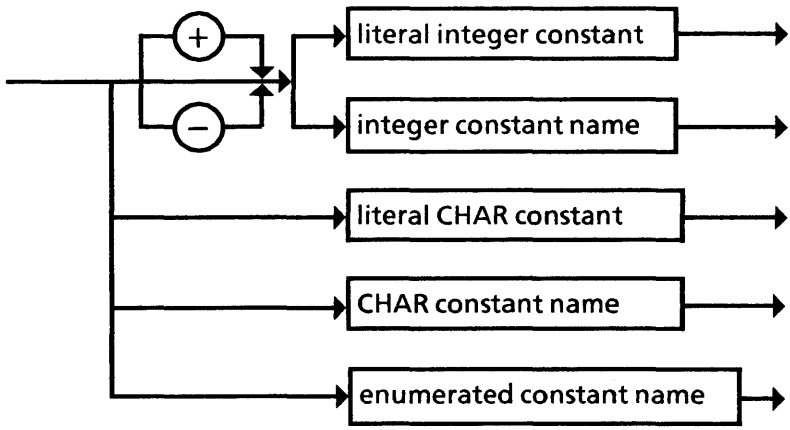


Figure 4-4. Limited Ordinal Constant Syntax

Initializers

Initializers supply constant initial values for variables and constant default values for optional value parameters. To specify an initializer, follow the item's type with the symbol ":=" (the *assignment operator*), followed by the initializer.

For example, the following VAR section shows several kinds of nonaggregate initializers:

```
VAR
   a: INTEGER : = 3;
   s: SET OF CHAR : = ['a'..'z','A'];
   s2: SET OF (red,yellow) : = []; { Empty set. }
   counter: INTEGER : = 1 + 2;
   c: [READONLY] REAL : = 3e10;
   p: ↑INTEGER : = NIL;
   str: STRING(80) : = ''; { Empty string. }
```

The following procedure heading illustrates an initializer used to supply a constant default value for a value parameter:

```
PROCEDURE p (c: CHAR : = 'A');
```

Here, for the procedure p, the default value for the value parameter c is the character A.

The syntax of initializers is shown in Figure 4-5.

Constant Initializers

In constant initializers, the constant must be assignment compatible with the target data item, and it is converted to the exact target type by the general rules for assignment. Strings are truncated or padded by spaces as required. If an ordinal constant is out of the target's range, the compiler detects the error.

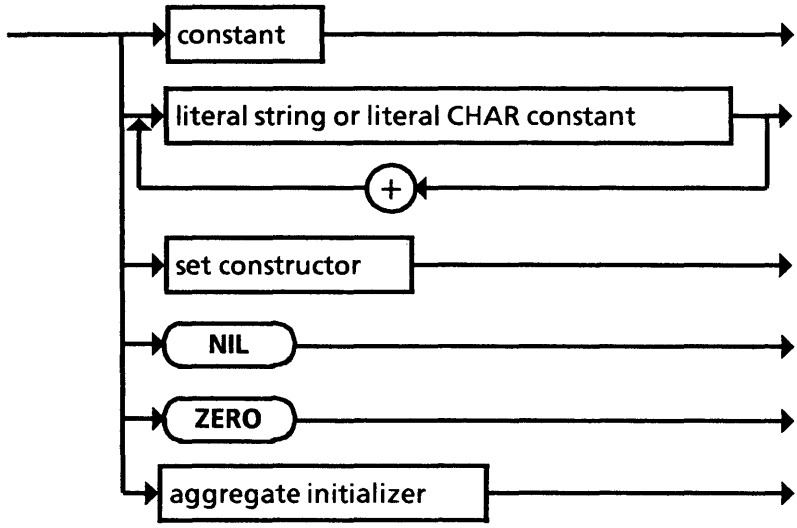


Figure 4-5. Initializer Syntax

As explained previously in the section "Constant Declarations," a constant can be an extent expression with constant operands. Note that a parenthesized extent expression is treated as an *aggregate initializer* (as defined later in this section) if the corresponding data item is an array or record.

Concatenated String Constants

The concatenation of literal string and/or literal CHAR constants provides a notation for expressing strings of more than 256 characters as initializers.

A string constant or concatenated string can be used to initialize a PACKED ARRAY[1..*n*] OF CHAR.

Set Initializers

Sets are initialized with set constructors enclosed in brackets ([]) that are either empty or consist of member designators separated by commas. Each member designator is either a constant extent expression or a pair of such expressions separated by the symbol '..'. A set constructor has the same meaning here as in an expression (see Chapter 6, "Expressions and Operators").

NIL

The reserved word NIL, which denotes the null pointer value, can be used only to initialize a pointer item.

ZERO

The ZERO function can be used in initializers. It is compatible with any data type and means that the entire storage of the variable is initialized to binary zero. For example, the following declaration initializes a record variable to binary zero:

```
VAR
   rec : RECORD
      .
      .
      END : = ZERO;
```

If used in an aggregate initializer for the tag of a variant part of a variant record, ZERO initializes the entire variant part to binary zero (in which case there must be no further initializers for the record).

In Pascal terms, the initialization with ZERO has the following meaning:

INTEGER	zero
REAL or DOUBLE	zero

BOOLEAN	FALSE
Enumerated	The enumerated element with ORD(element) = 0
CHAR	The character NUL
STRING	String of NUL characters
VARYING_STRING	The null string
Sets	The empty set
Pointer types	NIL

Note that an ordinal target with a subrange type can thus be initialized to a value outside the subrange. The compiler *does not* treat this as an error.

Aggregate Initializers

Aggregate initializers are used to initialize arrays or records, and an initial value must be specified for each element or field. The syntax is shown in Figure 4-6.

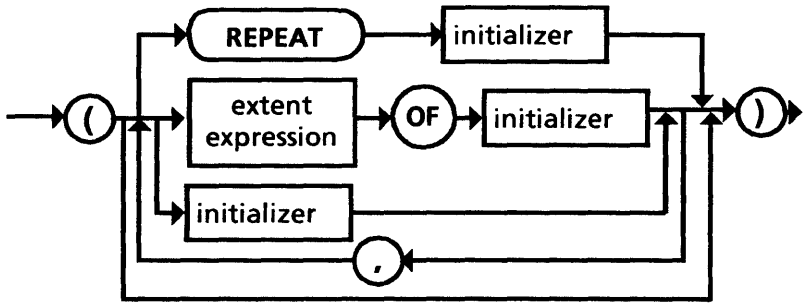


Figure 4-6. Aggregate Initializer Syntax

The OF and REPEAT constructions allow you to repeat an initial value for several fields or elements. The

extent expression preceding OF specifies the number of iterations. It must have constant operands and yield a nonnegative integer. REPEAT uses the subsequent initializer to fill out the remaining fields or elements in the data item. It's an error to explicitly specify too few or too many initializers, but REPEAT will correctly fill it out.

An empty aggregate initializer is valid only if the corresponding data item is an empty array or record. A parenthesized constant or extent expression can be used for an aggregate with one element.

In this context, multidimensional arrays are considered one-dimensional arrays with arrays as elements; nested aggregate initializers have to be used unless ZERO is used to zero the entire subarray.

For example, the following declaration initializes an array's elements to 1:

```
VAR
   a1: ARRAY[1..5,1..10] OF INTEGER
     := (5 OF (10 OF 1));
```

For variant records, there must be an initializer for the tag whether or not the record has an explicit tag field. Except where ZERO is used, the initializer for the tag must be compatible with the tag's type; the remaining initializers then match the selected variant. For example:

```
VAR
   x: RECORD
     CASE BOOLEAN OF
       true: (a: INTEGER);
       false: (b: STRING(8))
     END
       := (false, 'abc');
```

Here, field b is initialized to 'abc'.

Effects of Initializers

Initializers define constant values and are allowed only with constant-sized variables and parameters. For variables with the READONLY or VALUE attribute, the initializer simply defines the data item's constant value; no code needs to be executed.

For outer-level nonconstant variables, the data item is initialized by the system as part of job initialization. For local variables (VAR declarations within a PROGRAM or other routine), the data item is initialized by prologue code; that is, code that is executed just before the code for the routine's BEGIN–END.

An initializer for a value parameter supplies a default value that is used whenever an explicit argument is not supplied for the parameter. (See Chapter 8, "Procedures and Functions," for more information.)

In some cases, the use of an initializer is far more expensive in code space and/or execution time than would appear from looking at the source code. The following restrictions are imposed by the compiler to avoid some of the less efficient cases:

- No initialized data item can have a storage size greater than 65,535 bytes.
- If the initialized data item is a local (not outer-level) variable or a value parameter, at least one of the following must be true:
  - It has the READONLY or VALUE attribute.
  - It is initialized with the ZERO function.
  - It is initialized to a string with no more than 256 characters.
  - Its size does not exceed 256 bytes.

You should also bear in mind that initializing a large outer-level variable (unless it has the READONLY attribute) may involve moving a substantial amount of data at run time and require additional storage to hold the constant image of the initialized storage. This data movement may be less efficient than explicit assignments. For example, if you have a large table (array) with relatively few nonzero initial entries, it is usually more efficient to initialize the entire table with ZERO and then use assignment statements to supply the nonzero values.

Predeclared Named Constants

VAXELN Pascal provides the following predeclared named constants:

- The words TRUE and FALSE, which represent the two values of the type BOOLEAN.
- The word MAXINT, which represents the value 2,147,483,647; that is, $2^{31}-1$. This is the maximum value of type INTEGER. The minimum value of type INTEGER is $-(\text{MAXINT} + 1)$; that is, -2^{31}.
- The constant ASSERT_CHECK_ENABLED, which can be used to make the execution of code depend on the presence of the compiler qualifier CHECK=ASSERT. This qualifier enables assertion checking. If the qualifier is present, the constant is TRUE; otherwise, it is FALSE.

Predeclared Enumerated Types

In addition to the predeclared named constants described above, VAXELN Pascal provides the

following predeclared enumerated types, which in turn define predeclared named constants:

- The type EVENT_STATE is used in calls to CREATE_EVENT. The defined values are EVENT$CLEARED and EVENT$SIGNALED.
- The type FILE_ACCESS is used in calls to OPEN. The defined values are ACCESS$SEQUENTIAL and ACCESS$DIRECT.
- The type FILE_CARRIAGE_CONTROL is used in calls to OPEN. The defined values are CARRIAGE$LIST, CARRIAGE$FORTRAN, and CARRIAGE$NONE.
- The type FILE_DISPOSITION is used in calls to OPEN. The values are DISPOSITION$SAVE and DISPOSITION$DELETE.
- The type FILE_HISTORY is used in calls to OPEN. The defined values are HISTORY$OLD, HISTORY$NEW, HISTORY$UNKNOWN, and HISTORY$READONLY.
- The type FILE_RECORD_TYPE is used in calls to OPEN. The defined values are RECORD$FIXED and RECORD$VARIABLE.
- The type FILE_SHARING is used in calls to OPEN. The defined values are SHARE$NONE, SHARE$READONLY, SHARE$READWRITE.
- The type NAME_TABLE is used in calls to CREATE_NAME and TRANSLATE_NAME. The defined values are NAME$LOCAL, NAME$UNIVERSAL, NAME$BOTH (allowed only for TRANSLATE_NAME).
- The type OPEN_CIRCUIT is used in calls to OPEN. The values are CIRCUIT$CONNECT and CIRCUIT$ACCEPT.

- The type QUEUE_POSITION is used in calls to REMOVE_ENTRY and INSERT_ENTRY. The defined values are QUEUE$HEAD, QUEUE$TAIL, and QUEUE$CURRENT (allowed only for REMOVE_ENTRY).

Chapter 5
Variables

Introduction

VAXELN Pascal provides for the introduction of several sorts of variables; that is, items to which different values can be assigned during the execution of a program. This chapter discusses the declaration of variables, the rules for variable references, storage allocation, and data sharing between processes in a job.

A *local variable* is defined by use of a variable declaration (without the READONLY attribute) within the body of a routine. A value parameter declared without READONLY is also a local variable of its routine, the difference being that it is initialized to the corresponding argument value. In each invocation of a routine, a local variable is a distinct data item. Due to recursive use of a procedure or function or multiple parallel invocations of a process block, several instances of a local variable may exist at the same time.

An outer-level variable declaration without READONLY defines a single variable data item that exists for the entire duration of the job. An outer-level variable declaration may also specify the EXTERNAL attribute, which means that the data item is actually defined in a module not written in VAXELN Pascal.

A variable declaration with the READONLY attribute defines a constant data item. A special form of constant data related to the VAX/VMS linker is provided by a variable declaration with the VALUE attribute.

The construction used to reference a variable is called a *variable reference,* which can denote all or part of a variable, as follows:

- An indexed variable reference denotes an element in an array.
- A field reference denotes a field in a record.
- A pseudo variable reference calls the SUBSTR or ARGUMENT function to form a variable reference.
- An indirect variable reference denotes a data item specified by its address. Such an item can be introduced by a variable declaration, but it is usually an item allocated by use of the dynamic allocation routines discussed in the section "Storage Allocation."
- A buffer variable reference denotes the buffer associated with a file variable.
- A typecast variable reference references a data item using a data type that is different from that of the original reference.

Special care must be taken in referencing data that is shared between processes. The last section of this chapter, "Interprocess Data Sharing," discusses data sharing within a single job. Sharing data between jobs is discussed in Chapter 12, "Interjob Communication."

Variable Declarations

The syntax for variable declarations is shown in Figure 5-1.

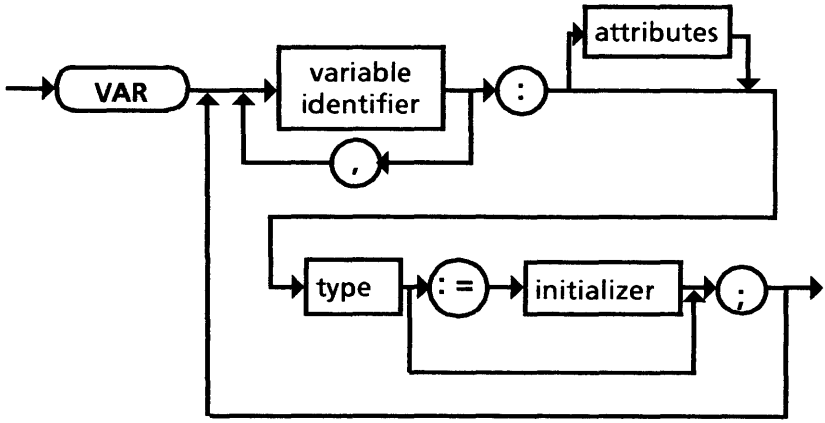


Figure 5-1. Variable Declaration Syntax

Each variable identifier in the list is declared as the name of a variable. The scope of the declaration is the containing routine body. All variables in the list have the type, attributes, and initializer specified on the right-hand side of the colon (:).

The READONLY, VALUE, and EXTERNAL attributes apply specifically to variables and are described in the following subsections. The ALIGNED, BIT, BYTE, WORD, and LONG attributes may also be used, depending on the specified type. These attributes are described under "Data Representation" in Chapter 3.

The initializer supplies a constant initial value for the variable (or variables). An initializer is required for READONLY and VALUE variables without the EXTERNAL attribute; it is not allowed with EXTERNAL. Other rules for initializers depend on the variable's data type, as explained under "Initializers" in Chapter 4.

READONLY Attribute for Variables

The READONLY attribute, as used in a variable declaration, specifies that the variable is to be allocated in readonly storage so that it is a form of constant data. The compiler issues an error message for any explicit attempt to modify a READONLY variable. Modifications not detectable by the compiler (for example, via a pointer to the variable) result in an access violation exception at run time.

Unless EXTERNAL is specified, a READONLY variable must have an initializer, since there is no other way to give it a value. The variable must not be a file or contain a file.

The actual memory allocation of EXTERNAL READONLY variables depends on the external definition, so the compiler cannot guarantee write protection at run time. However, it does issue the usual error message for any explicit attempt to modify the variable.

Note that the READONLY attribute can also be specified for value parameters, as explained in Chapter 8, "Procedures and Functions."

VALUE Attribute

The VALUE attribute is used in a variable declaration to define a form of constant data item related to the VAX/VMS linker. If the variable's name is exported without EXTERNAL, the defined value is made available to the linker as the value of a global symbol with the same name. If the declaration also specifies EXTERNAL, the name denotes a value that will be supplied to the linker by a non-VAXELN Pascal module.

The following rules apply to variables declared with the VALUE attribute:

- Unless the EXTERNAL attribute is also specified, the declaration must contain an initializer. This determines the value denoted by the name.
- A variable reference to the declared item must be its name only (typecasting, for example, is not allowed) and it must occur in a context denoting the item's value (that is, as a function in an expression).
- The data type of the variable must an ordinal type, REAL, a small set type (that is, if ORD(maximum-element) ≤ 31), or a pointer type. It must not have the BIT attribute. All these types require ≤ 32 bits of storage. The value made available to the linker is the appropriate 32-bit representation; for example, a value of type CHAR would be zero extended to 32 bits.

Note that VALUE can be used for non-exported, non-EXTERNAL variables, but this is not a significant usage.

EXTERNAL Attribute

The EXTERNAL attribute is used in a variable declaration to indicate that the variable is actually defined in a non-VAXELN Pascal module. If the declaration has the VALUE attribute, the item must be available to the VAX/VMS linker as a global value. Otherwise, it must be available as a global symbol denoting the address of storage defined by another module.

Variable References

Figure 5-2 shows the various forms of variable references.

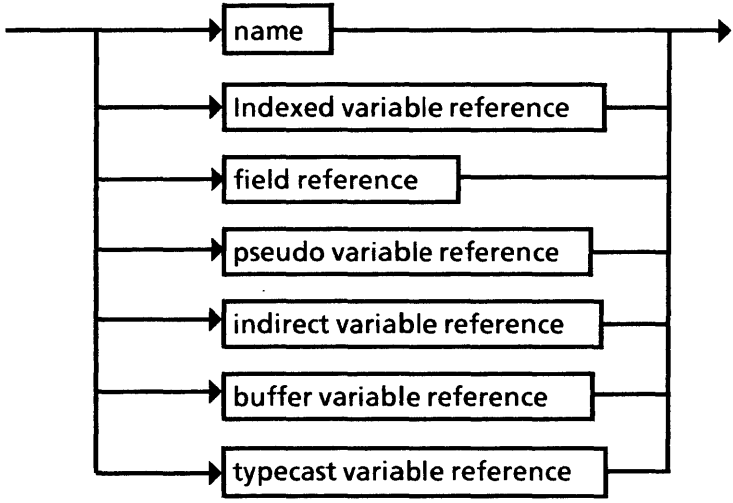


Figure 5-2. Variable Reference Syntax

The name specified in a variable reference can be the name of a variable (VAR declaration), value parameter, VAR parameter, a name established as a variable name by a WITH statement, or a function name (left-hand side of assignment statement within the function).

If the reference is simply a name, it references the entire data item denoted by the name, and the reference's data type is the same as the name's type.

The other forms of variable reference restrict the reference to part of a data item, access a related item, or

redefine the assumed type of the item. These forms are explained in the following subsections.

Indexed Variable References

An *indexed variable reference* selects an element of the array denoted by the initial variable reference shown in Figure 5-3.

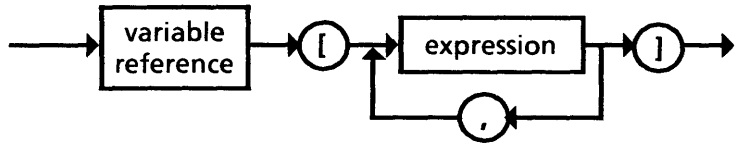


Figure 5-3. Indexed Variable Reference

Each expression provides a subscript value. The expression's type must be assignment compatible with the corresponding subscript type (index range) in the array's type definition. If the array has more than one dimension, multiple expressions can be written, separated by commas, in the same set of brackets. It is a range violation if the expression's value is outside the corresponding index range.

The data type of the indexed variable reference is the array's element type. If the array is multidimensional and not fully subscripted in this reference, the indexed variable reference will have an array type.

Examples:

```
VAR
   namearray: ARRAY[1..10] OF
   VARYING_STRING(80);
   stats:RECORD
```

```
        filenum: INTEGER;
        table: ARRAY [1..4,1..6] OF REAL;
      END;
BEGIN
  namearray[1] : = 'Shakespeare';
  stats.table[1][1] : = 1.239e10;
  stats.table[1,2] : = 5.0e21;
  stats.table[1] : = stats.table[2] ;
END.
```

Here, namearray, stats.table[1], and stats.table all are references to arrays. That is, namearray and stats.table refer to entire arrays, and stats.table[1] refers to an array element that is itself an array. Notice, therefore, that an element in the multidimensional array stats.table can be referred to either with a sequence of bracketed indices or with a sequence of indices in the same brackets.

Field References

A *field reference* selects a field in the record denoted by the initial variable reference shown in Figure 5-4.

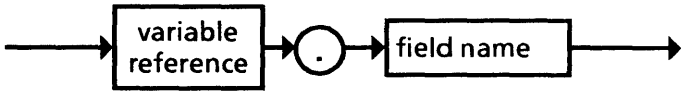


Figure 5-4. Field Reference

The field name must be one of the field names defined in the original definition of the record type. The data type of the field reference is the same as the type of the field within the record type.

Examples:

```
TYPE rtype = RECORD
   person : VARYING_STRING(80);
   stats : RECORD
     salary : REAL;
     hiredate : LARGE_INTEGER;
     END { stats. }
   END; { rtype. }
VAR
   r : rtype;
   rptr : ↑rtype;
   rarray : ARRAY [1..10] OF rtype;
BEGIN
   r.person : = 'Arturo Toscanini';
   r.stats.salary : = 15000.00;
   r.stats.hiredate : = TIME_VALUE(
     '1-JAN-1939 00:00');
   NEW(rptr);
   rptr↑ : = r;
   rarray[1].stats.hiredate : = rptr↑.stats.hiredate;
END;
```

Here, r, r.stats, rptr↑, and rarray[1] all are references to records and so can be followed by field names to designate specific fields. Used by themselves, they refer to entire records, as in

```
rptr↑ : = r;
```

which assigns the entire contents of record r to the record identified by rptr.

Note that in a reference to a field in a nested record, there must be a field selection for each level of record nesting. Therefore, in the above example, the field hiredate in r must be referenced as r.stats.hiredate. The compiler will reject r.hiredate because hiredate is not the name of a field immediately in r.

The WITH statement provides a method to temporarily establish the field names in a record as variable names, so they can be used without further qualification. For example:

```
WITH r.stats DO
   salary : = salary + 1;
```

Here, the WITH statement establishes a reference to the record r.stats. Within the statement's body, this reference applies to the field name salary, even though it is not preceded by a record reference. The WITH statement is described in Chapter 7, "Pascal Statements."

Pseudo Variable References

The predeclared functions SUBSTR or ARGUMENT can be used to form a *pseudo variable reference.* The syntax is shown in Figure 5-5.

Figure 5-5. Pseudo Variable Reference

The function call is an invocation of the SUBSTR function with a variable reference as its first argument, or the ARGUMENT function with a VAR parameter as its first argument. These functions are described in Chapter 9, "VAXELN Routines."

Indirect Variable References

An *indirect variable reference* denotes the data item whose address is given by the value of the initial

variable reference shown in Figure 5-6. This reference must have a pointer data type.

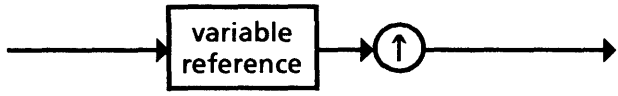


Figure 5-6. Indirect Variable Reference

The data type of the indirect reference is the same as the associated data type of the pointer. The indirection character is either a caret (^), up-arrow (↑), or *at* sign (@). The up-arrow is generally used in this manual, for clarity, but the *at* sign and caret are more commonly available on terminal keyboards.

The following example shows the use of pointers to access fields in a dynamically allocated record:

```
TYPE
   gender = (male, female);
   persrec = RECORD
     name: VARYING_STRING(80);
     age: 0..200;
     CASE sex: gender OF
     male:
     (beard: BOOLEAN;
     bdate: LARGE_INTEGER);
     female:
     (births: BOOLEAN;
     birthdate: LARGE_INTEGER);
     END;
VAR
   patient: ↑persrec;
```

```
BEGIN
  NEW(patient);
  WITH patient↑ DO BEGIN
    name : = 'Melvin Cowsnofski';
    age : = 45;
    sex : = male;
    beard : = TRUE;
    bdate : = TIME_VALUE('29-FEB-1956')
    END
END.
```

Here, patient↑ refers to an entire record allocated by the NEW procedure, and a reference to the record is established by the WITH statement.

Buffer Variable References

A *buffer variable reference* denotes the buffer associated with the Pascal file variable specified by the value of the initial variable reference shown in Figure 5-7. This reference must have a file data type.

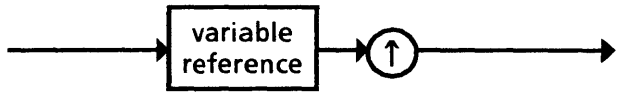


Figure 5-7. Buffer Variable Reference

The data type of the buffer reference is the same as the file type's component type. The indirection character is either a caret (^), up-arrow (↑), or *at* sign (@). The up-arrow is generally used in this manual, for clarity, but the *at* sign and caret are more commonly available on terminal keyboards.

For example,

```
output↑ := 'a';
```

assigns the character 'a' to the buffer of the predeclared file output. The type of the reference output↑ is CHAR.

Note that a reference to a file's buffer or the value in its buffer is not always valid and may trigger actual I/O, as explained in Chapter 15, "Input and Output."

Typecast Variable References

A *typecast variable reference* is used to reference a data item using a data type that is different from that of the original reference. The new type is specified by the named type in the typecast variable reference, as shown in Figure 5-8.

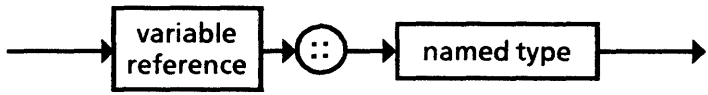


Figure 5-8. Typecast Variable Reference

The named type can be a type name, a bound flexible type, or a pointer to one of these types. (See Chapter 3, "Data Types," for the exact named type syntax.)

In general, these types have the same meaning as when used in a declaration. However, there are two special cases of interest:

1. Typecasting to BYTE_DATA.

   The type name BYTE_DATA can be used as a complete named type. In this case, it is interpreted

as BYTE_DATA(*n*), where *n* is the storage size of the original variable reference. For example:

```
VAR x: DOUBLE; y: LARGE_INTEGER;
x::BYTE_DATA := y::BYTE_DATA
```

Here, both typecast references have the type BYTE_DATA(8), and the assignment statement simply moves the contents of y to x.

2. Typecasting to a flexible type.

   The new type of a typecast reference may be specified as a bound flexible type or as a pointer to such a type. In either case, the extents of the bound flexible type may be given as arbitrary expressions. They are not restricted to the special class of extent expressions. The only requirement is that the extent expression be assignment compatible with the corresponding extent parameter of the flexible type.

   In typecasting to a bound flexible type, the expressions are evaluated each time the typecast reference is evaluated. This provides a completely general way to describe dynamically sized data. (For examples, see the discussion of the WITH statement in Chapter 7, "Pascal Statements.")

The size of the new data type in a typecast variable reference must not exceed the size of the original type (to avoid a range violation error). In addition, if the reference is not addressable, the new data type must have a boundary requirement of bit alignment. (Other data types are not consistent with non-addressability of data.)

A typecast variable reference denotes the data item at the same location as the original reference, extending for the size specified by the new type. Typecasting has no effect on the contents of the data item. The effects of

using a typecast reference are dependent upon the internal representation of data, and the construction is not part of standard Pascal.

Incorrect use of typecasting leads (immediately or later) to an attempt to use an invalid value, because the contents of the data item are inconsistent with the applicable data type. Note that typecasting a literal or named constant is not allowed. You must use a READONLY variable initialized to the constant value; this clearly defines the internal representation of the constant.

The following example shows the use of typecasting to examine the individual parts of a REAL data item:

```
TYPE f_float = PACKED RECORD
   { Hidden bit is not in the internal representation. }
   fraction1: 0..127;      { 7 bits unsigned. }
   exponent: 0..255;       { 8 bits unsigned, biased. }
   sign      : BOOLEAN;{ 1 bit. }
   fraction2: 0..65535;  { 16 bits unsigned. }
   END;

VAR m: INTEGER;  x: REAL;

BEGIN
   x := - 1.25e10;
   WRITELN(hex(x));  { Prints 43B7D13A. }
   WITH x::f_float DO
   BEGIN
     { Print unbiased exponent as decimal integer. }
     WRITELN(exponent - 128); { Prints 34. }
     { Print fraction including hidden bit as a hex
       integer. }
     m := (128 {Gives hidden bit. } +
       fraction1)*65536 + fraction2;
     WRITELN(hex(m));  { Prints BA43B7. }
```

```
    { Produce a reserved operand value. }
    exponent : = 0;
    sign : = TRUE;
    WRITELN(hex(x));    { O.K., x accessed as
      BYTE_DATA. }  { Prints 43B7803A. }
    m : = TRUNC(x); { Reserved operand exception. }
    WRITELN(m);
  END;
END;
```

This example is typical of typecasting in that more than one reference to a typecast item (x::f_float) is needed. The WITH statement is useful in this situation; see the discussion of the WITH statement in Chapter 7, "Pascal Statements," for examples of using the construction "WITH name AS" to handle more general typecasting.

The possibility of typecasting a packed field occurs often; however, care must be taken in order to get the representation exactly right. The following example illustrates the typecasting of packed fields:

```
VAR x:  PACKED RECORD   { 16 bits. }
  a : 0..127;           { 7 bits unsigned. }
  b : PACKED SET OF 0..7; { 8 bits. }  { Has 7-bit
    offset, so is not addressable. }
  c : BOOLEAN;
  END;

TYPE  t7 = [bit(7)] 0..127;
      t8 = [bit(8)] 0..127;
      t9 = [bit(9)] 0..127;

BEGIN
  x.b : = [0..7];  { Sets all 8 bits of x.b. }
  WRITELN(x.b::CHAR); { The compiler issues an
      error message rejecting typecasting because
      type CHAR cannot apply to a non-addressable
      reference. }
```

```
    WRITELN(x.b::t7); {Allowed. Access on the first 7
        bits of the 8-bit field, so 127 is printed. }
    WRITELN(x.b::t8); {Allowed. Treats the field as an
        8-bit integer with the high-order bit as the sign
        bit. Because the subrange specification is
        0..127, the sign bit should be zero. The effect is
        to print – 127 without any indication of an
        error. }
    WRITELN(x.b::t9); { The compiler issues a warning
        message detecting a range violation because
        the type being cast to is larger than the
        constant size of the variable being cast.
        Executing the program results in an exception
        referencing the specific error. }
END;
```

Addressability of Variable References

A variable reference may denote a data item that is not addressable; that is, it is not located at a byte boundary in the VAX–11 memory. Such a reference may not be used as the argument of the ADDRESS function or as an argument passed to a VAR parameter in any routine. The compiler issues an error message for any attempt to use a non-addressable reference in these contexts.

The compiler considers a reference to be non-addressable only if both of the following are true:

- The reference's data type has a boundary requirement of bit alignment.
- The reference has a bit offset (from an addressable location), excluding the case of a constant bit offset divisible by eight.

Storage Allocation

This section summarizes the way the compiler allocates storage for constants and variables.

If a READONLY variable or a named string constant is exported from a module, it is allocated storage in the code program section (PSECT) of that module. Otherwise, constants, READONLY variables, and VALUE variables are allocated storage in the code PSECT of that module only as required by the generated code in a module using the item. The code PSECT is potentially shared by all instances of the program.

An outer-level, non-READONLY variable is allocated in the data PSECT of the containing module if it is exported or if it is required by the generated code. (Thus, unreferenced nonexported variables are not allocated storage.) The data PSECT is materialized separately for each instance of the program (as a job), which requires copying any nonzero initializers for the data.

In a VAXELN job, there is one copy of the data PSECT shared by all processes in the job. Thus, any data in this PSECT is potentially shared by the processes in a job.

Storage allocation for a local variable depends on its data type and how it is used by the generated code. Unreferenced constant-sized variables (including those eliminated by optimization) are not allocated storage. If a constant-sized variable is used in the generated code, it will be assigned to one or more registers and/or a location in the routine's stack frame, depending on its data type and pattern of usage. In any case, the allocation is, at most, for the duration of the routine's activation. Passing a variable to a VAR parameter,

referencing it from a nested routine, or taking its address will force allocation at a stack frame location.

Dynamically sized local variables are allocated by extending the routine's stack frame in prolog code that is executed immediately before the code for the routine's executable statements (BEGIN ... END).

The preceding discussion of local variables applies to variables declared in non-in-line routines. In each expansion of an in-line routine, a constant-sized local variable is treated as though it belonged to the non-in-line invoking routine. If required, it will be allocated in registers and/or the invoking routine's stack frame. Local variables of disjoint in-line expansions may share storage in the stack frame.

Dynamically sized local variables for in-line routines are allocated on the stack by prolog code generated as part of each in-line expansion of the routine. They are temporary additions to the invoking routine's stack frame and will be deallocated with it if not before. (For more information on VAX memory management and the VAX stack architecture, see the *VAXELN User's Guide*.)

Interprocess Data Sharing

Data may be shared (that is, accessed) by more than one process in a job. All data is potentially shareable except local variables of routines and value parameters of routines. These data items are allocated by the compiler in P1 memory (stack) space, and the address is meaningful only within the process allocating the data, since the stack space is private to each process in a VAXELN job.

Outer-level variables can be shared by name; that is, more than one process can refer to the variable by its

name. Sharing can also be accomplished with pointers and with VAR parameters of process blocks.

Sharing constant data, including variables declared with READONLY or VALUE, presents no programming problems. However, sharing data that is modified by one or more processes must be carefully managed to prevent unpredictable program behavior.

The following atomic operations can be used safely on data shared by processes within a job:

- The READ_REGISTER and WRITE_REGISTER routines. (Note that they are not restricted to operations on actual device registers.)
- The procedures INSERT_ENTRY and REMOVE_ENTRY, when used on the head and tail entries of a queue.
- The ADD_INTERLOCKED function.

If more complicated operations are performed on shared data, the access to the data must be synchronized. While one process is executing code that can modify the data, no other process can execute code that can access the data in any way. The synchronization must be done with kernel procedures or with the mutex routines (which call the kernel procedures themselves when necessary), as discussed in Chapter 11, "Subprocesses and Synchronization."

Failure to observe this principle results in unpredictable program behavior. A program that works on one processor model can fail on another, or a change to the VAXELN system might cause program failure.

Notes

The following notes supply additional guidelines regarding shared data.

Dynamic variables. Data allocated by NEW, like most other data, can be shared. The operations NEW and DISPOSE are atomic.

File variables. File variables are subject to the same rules as other data, and almost any operation on a file variable is a modify operation. Failure to synchronize the access to a file can result in scrambled input or output data or in a run-time error (if the Pascal run-time routines detect simultaneous access).

Initialization of shared data. It is good practice to initialize shared outer-level variables in the master process before subprocesses are created. It is easy to forget that the initialization operation must be synchronized; for example, initialization of a queue by START_QUEUE or of a MUTEX variable by CREATE_MUTEX.

Record locking. Programs that use shared data often must protect data more complicated than single Pascal variables. For example, if multiple processes are updating records in a File Service file using a single Pascal file variable, they must synchronize access to the file variable, but they must also protect (or "lock") records in the file. Otherwise, two read–rewrite sequences on the same record can get interleaved.

Shared messages. A message and its associated "text" variable can be manipulated by more than one process in a job, but the operations must be properly synchronized. For example, if process A deletes a message while process B is preparing to send it, the program will misbehave. Process B may get the status KER$_BAD_VALUE from SEND, or it may get an exception when it tries to access the message's text variable; these are relatively harmless results. It is also possible for process B to access some new, unrelated data via the address of the original text variable or

even (rarely) for the 32-bit MESSAGE value to be reused as a new object ID before B sends the message.

Communication regions. The communication region of an interrupt service routine is shared in a special way between the interrupt service routine and the rest of the device driver program. The program logic of the device driver must ensure that nonatomic operations are synchronized.

Device registers. Device registers are not shared data in the sense used above. In some cases, all they do is symbolize the responses of a device to events, such as read and write requests, that occur on the bus. The only predictable operations on device registers are READ_REGISTER and WRITE_REGISTER.

Chapter 6
Expressions and Operators

An *expression* in VAXELN Pascal represents the value of a constant, a variable, or a function result, or it may represent a combination of two or more such values separated by operators, or a range of such values.

VAXELN Pascal provides the following classes of *operators*:

- Arithmetic operators
- Boolean operators
- Relational operators
- Set operators
- Concatenation operator

This chapter discusses the syntax of Pascal expressions, including a discussion of operator precedence and associativity, followed by a discussion of side effects in expressions. The remainder of the chapter discusses each of the classes of operators individually.

Expression Syntax

In the syntax for VAXELN Pascal expressions, an expression consists of a *simple expression*, or two simple expressions separated by an operator. Each simple expression contains one or more *terms*, each consisting of a single *factor*, or two or more factors separated by operators. Each factor yields a value.

This summary of the terminology used in expression syntax is expanded in the following subsections.

Expressions

An *expression* in terms of the VAXELN Pascal syntax consists of a simple expression, or two simple expressions separated by a relational operator.

The expression syntax is shown in Figure 6-1.

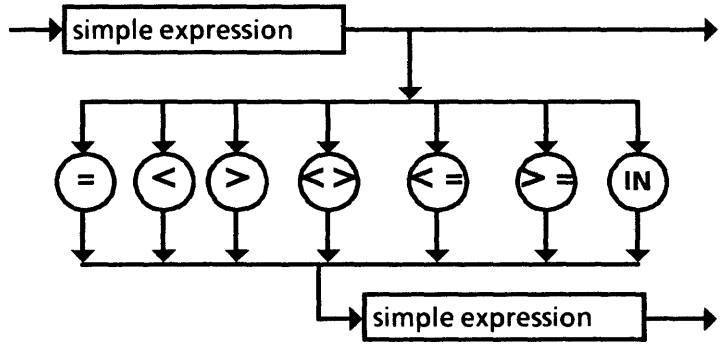


Figure 6-1. Expression Syntax

The relational operators shown in the syntax diagram are discussed in the section "Relational Operators," later in this chapter.

The following is an example of a VAXELN Pascal expression:

```
-a/b + a*d < a/b + a*d
```

Simple Expressions

A *simple expression* in the expression syntax contains one term, possibly preceded by the monadic + or monadic − operator, or two terms separated by one of a specific set of dyadic operators.

The syntax for a simple expression is shown in Figure 6-2.

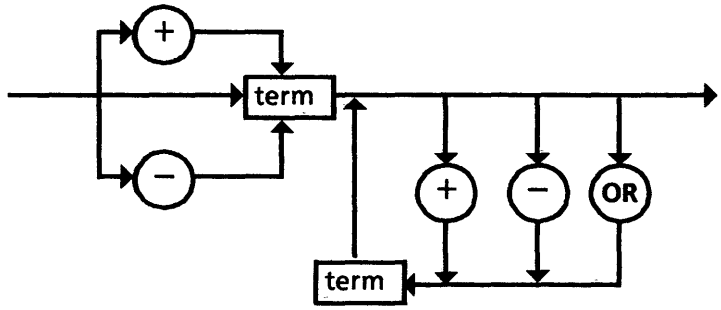


Figure 6-2. Simple Expression Syntax

The + and − operators shown in the syntax diagram are discussed in the section "Arithmetic Operators," later in this chapter. The operator OR is discussed under "Boolean Operators."

The following is an example of a simple expression:

```
-a/b + a*d
```

Terms

Each *term* in the simple expression syntax consists of a single factor, or two or more factors separated by one of a specific set of dyadic operators.

The term syntax is shown in Figure 6-3.

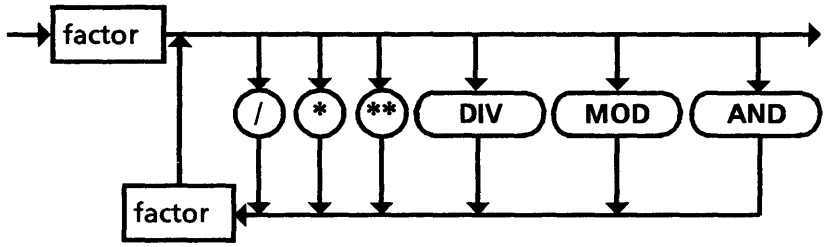


Figure 6-3. Term Syntax

The /, *, **, DIV, and MOD operators shown in the syntax diagram are discussed in the section "Arithmetic Operators," later in this chapter. The operator AND is discussed under "Boolean Operators."

The following are examples of terms:

```
a/b and a*d
```

Factors

Each *factor* in the term syntax is a primary expression that yields a value. A factor can be a literal constant, a constant name, a variable reference, a function call, an expression enclosed in parentheses, or a set constructor. (Set constructors are discussed later in this chapter; the other types of factors are discussed in detail elsewhere in this manual.)

In addition, the reserved word NIL can be used as a factor, and the logical operator NOT can be applied to a factor. (NOT is discussed in the section "Boolean Operators," later in this chapter.)

The factor syntax is shown in Figure 6-4.

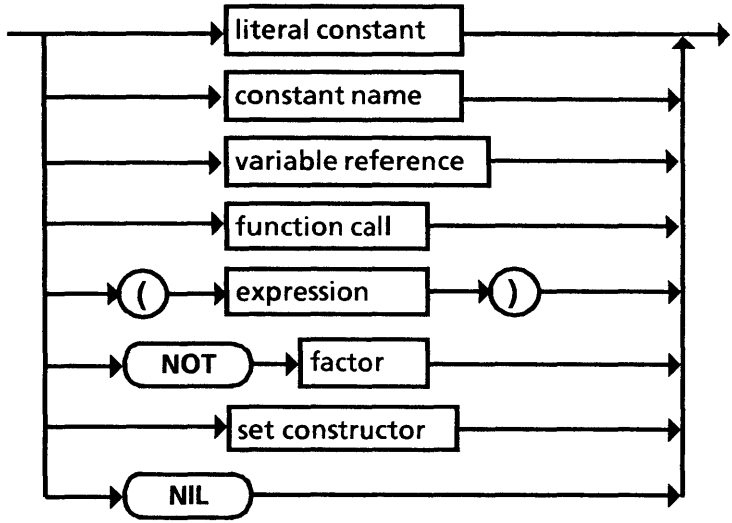


Figure 6-4. Factor Syntax

Operator Precedence and Associativity

When an expression contains several different operators, the rules for operator *precedence* define the grouping of operands with operators. The *associativity* rules define the evaluation of operations with equal precedence.

The rules for precedence and associativity are needed by compilers to produce consistent results. Although they tend to produce what you expect in simple cases, they are not meant to be memorized and relied upon in programming. Instead, it is usually more practical to use parentheses to eliminate ambiguity.

Precedence

Any expression can be enclosed in parentheses; subexpressions in the deepest level of parentheses are evaluated first. Within each expression or subexpression, the precedence of operators is as follows:

1. The exponentiation operator (**) has the highest precedence.
2. The exponentiation operator is followed by the monadic operators (NOT, +, and −).
3. The monadic operators are followed by the multiplicative operators (*, /, DIV, MOD, and AND).
4. The multiplicative operators are followed by the additive operators (+, −, and OR) and the string concatenation operator (+).
5. At the lowest level of precedence are the relational operators (=, <>, <, >, <=, >=, and IN).

For example, the expression

```
-2.0**2
```

is equivalent to

```
-(2.0**2)
```

giving the result -4.0.

Parentheses, and nested parentheses, can be used to treat subexpressions as single operands. For example,

```
NOT A OR B
```

means "the OR (union) of NOT A with B," because NOT has higher precedence than OR. However,

```
NOT (A OR B)
```

means "the logical negation of the union of A with B," because (A OR B) is treated as a single operand.

Associativity

The associativity of all dyadic operators of equal precedence is from left to right. For example,

```
A/B*C
```

is equivalent to

```
(A/B)*C
```

Likewise, the expression

```
x**y**z
```

is equivalent to

```
(x**y)**z
```

Usually, the associativity rules produce the result you would expect from your knowledge of arithmetic. However, you must sometimes introduce parentheses to prevent floating-point errors. For example, the parentheses in the following expression are required, to avoid a floating-point overflow:

```
3.0e10 *( 1.0e209*1.0e - 199)
```

If the parentheses are omitted, the first two factors are multiplied first, and their product (3×10^{309}) is too large to be a valid REAL or DOUBLE value (floating-point overflow). With the parentheses, the second and third factors are multiplied first, and their product (1×10^{10}) is a valid result.

Side Effects in Expressions

If evaluation of an expression's operands has side effects (beyond determining the operand's values), the program's behavior may be sensitive to the compiler's

code generation methods and may vary depending on such things as the compiler's version or the setting of command options such as /CHECK and /OPTIMIZE. Programmers should be aware of the following considerations.

The order of operand evaluation is unpredictable. Misinterpretation of this fact is a common source of programming mistakes. For instance, consider the following expression, in which F, G, and H are function calls:

```
F + G + H
```

The rules described previously specify *only* that the *sum* F + G is computed first, *not* that function F is called first nor, in fact, that the functions are called in any predictable order. In particular, parentheses change only the associativity and not the order of operand evaluation. That is, in the expression

```
F + (G + H)
```

there are still no guarantees about which function is called first, only that the sum G + H is computed first.

Obviously, then, a program containing such an expression must make no assumptions about the calling order, such as having one function modify a variable in preparation for use by another. Mistakes of this kind are best avoided initially, because they can be quite difficult to detect. For instance, if the functions really must be called in a certain order, call them in that order and then add up their results:

```
a := F; b := G; c := H; d := a + b + c;
```

Similarly, the arguments in a procedure or function call are not evaluated in any predictable order, and the same caution applies. The caution also applies to

expressions used as subscripts in a reference to an element of a multidimensional array.

Note that the unpredictability of the order of evaluation applies across an entire simple statement. Thus, for

```
x[k] : = f;
```

the value of k may be taken before or after the call to f. Try this with f modifying k and with /CHECK and /NOCHECK.

The evaluation of expressions involving AND and OR may cease as soon as the final result is obvious. An optimization of this kind (so-called "short-circuit evaluation") can save significant execution time when logical expressions are long, but as a result, you should avoid certain assumptions when writing programs.

For example:

```
IF A AND F(I) ...
```

Here, evaluation stops immediately if either operand is FALSE, since the result is then certainly FALSE. Once again, this may lead to programming mistakes when, for example, F is a function with some effect other than simply returning a BOOLEAN value (a so-called "side-effect")—such as calling another function or procedure, modifying a global variable, and so on. If A happens to be evaluated first and is FALSE, function F will not be called, which might change the behavior of the program. The only precaution is to avoid these constructions altogether if one of the operands is a function with side effects. For example:

```
B : = F(I);
IF A AND B...
```

A similar caution applies to expressions involving the OR operator, the evaluation of which may stop as soon as an operand is TRUE.

It is also possible to make mistakes by *depending* on short-circuit evaluation, since all logical operands are evaluated if you do not compile the program with optimization. For example:

```
IF pointer <> NIL AND pointer↑ > 0 THEN ...
```

Here, an error occurs if you compile with NOOPTIMIZE, because when pointer does equal NIL, pointer↑ is an illegal reference.

When a program behaves differently with the OPTIMIZE and NOOPTIMIZE compiler qualifiers, short-circuit evaluation is a likely cause. (See Chapter 16, "Program Development," for more information on compiling and compiler qualifiers.)

Arithmetic Operators

The arithmetic operators are shown in Table 6-1. The operands must have arithmetic data types, such as INTEGER and REAL.

The arithmetic operators produce the result that is defined by the rules of arithmetic. If the result is too large to be represented by the result type, an overflow error occurs, and the result is undefined. If the result of a floating-point operation is too small (in absolute value) to be represented, a floating-point underflow error occurs. (See "Overflow and Underflow," later in this section, for more information.)

Table 6-1. Arithmetic Operators

Operator	Operation	Operand Type	Result Type
Dyadic			
+	Addition	Integer[1], floating[2]	Integer or floating-point[3]
–	Subtraction	Integer[1], floating	See +
*	Multiplication	Integer[4], floating	See +
**	Exponentiation	Integer[5], floating	Floating-point[6]
/	Division	Integer[4], floating	Floating-point
DIV	Division	Integer[4]	Integer
MOD	Modulo	Integer[4]	Integer
Monadic			
–	Sign inversion	Integer[1], floating	Same as operand's
+	Identity	Integer[1], floating	Same as operand's

[1]Here, "integer" means type INTEGER or LARGE_INTEGER. The type LARGE_INTEGER can be combined only with integers.

[2]Here, "floating" means the types REAL and DOUBLE.

[3]If both operands are integers, the result is an integer; otherwise, the result is REAL or DOUBLE.

[4]The type LARGE_INTEGER is not allowed.

[5]The first operand must be REAL or DOUBLE; the second operand must be INTEGER, REAL, or DOUBLE.

[6]If either operand is DOUBLE, the result is DOUBLE; otherwise, the result is REAL.

Operands of Different Types

The monadic operators always produce results of the same type as the operand's. The rules for the result type with dyadic operators are as follows:

1. If either operand is of type LARGE_INTEGER, the other must be INTEGER or LARGE_INTEGER, and the result is of type LARGE_INTEGER.
2. Otherwise, if either operand is of type DOUBLE, the result is of type DOUBLE.
3. Otherwise, if either operand is of type REAL, the result is of type REAL.
4. Otherwise, both operands are of type INTEGER and the result is also of type INTEGER.

The effect of these rules is to perform computations and produce results with the same degree of precision as the most precise operand's.

Overflow and Underflow

If the result of a floating-point operation has a magnitude larger than the maximum allowed for the data type, the operation is said to *overflow*. The same applies if the magnitude of an intermediate result exceeds the allowed maximum. Overflow causes one of the exceptions SS$_FLTOVF or SS$_FLTOVF_F to be raised.

If the result of a floating-point operation is non-zero but has a magnitude smaller than the minimum allowed for the data type, the operation is said to *underflow*. The same applies if the magnitude of an intermediate result is less than the allowed minimum. The effect of underflow depends on whether or not the underflow exception is enabled. This can be controlled by the use

of the UNDERFLOW and NOUNDERFLOW attributes in routines, as discussed in Chapter 2.

By default, the underflow exception is disabled. In this case, underflow yields a floating-point zero result and no exception is raised. If the underflow exception is enabled, one of the exceptions SS$_FLTUND or SS$_FLTUND_F is raised.

Addition, Subtraction, Multiplication, Sign Inversion, Identity

These operations produce integers if both operands (or the sole operand) are integers; otherwise, the result is a floating-point value. The operations all follow the usual arithmetic rules, unless underflow or overflow occurs.

Exponentiation

Exponentiation (that is, x**y) always produces a REAL or DOUBLE result, since the first operand, x, must be REAL or DOUBLE. The case of x an integer is not allowed in VAXELN Pascal, although y can be an integer.

Division and DIV

Division (with the / operator) always produces a REAL or DOUBLE result, since, in general, two integers are not exactly divisible. Consequently, the result of the operation cannot be assigned to an integer variable.

The DIV operator always produces an integer, and both its operands must be integers. A DIV operation, in effect, produces a floating-point result and removes the fraction to produce an integer. For example, the result of the operation 1/2 is 0.5, while the result of the operation 1 DIV 2 is 0.

The divisor must not be zero in either case, or else one of the following exceptions is raised at run time: SS$_INTDIV, SS$_FLTDIV, or SS$_FLTDIV_F.

MOD

Given an integer I and a positive integer J (the *modulus*), the result of the operation

```
I MOD J
```

is the value, for some integer *k*, of

$$I - (k \times J)$$

such that

$$0 \leq I \text{ MOD } J < J$$

Note that J must be positive, not zero or negative. If I = J, or I is a multiple of J, or I = 0, then I MOD J is 0.

An elementary example of modulo arithmetic is "clock arithmetic," in which the modulus is 24. For example,

```
hour := advance MOD 24;
```

where the 24-hour clock (values from 0 to 23 hours, initially 0) is advanced by some number of hours given by advance (or, if advance is negative, the clock is turned back).

If advance is 25, hour is 1. If advance is 58, hour is 10. If advance is 0, the clock is not advanced, and hour is still 0. Finally, if advance is −25, then *k* is −2, and hour is 23. (That is, if the current time is 0, and you turn back the clock 25 hours, the new time is 23.) Note that:

$$-(I \text{ MOD } J) \neq -I \text{ MOD } J$$

Note also that I MOD J is the same as the remainder of I DIV J, but *only when I is greater than or equal to zero*; for example, the remainder of −25 divided by 24 is −1, which is not the same as −25 MOD 24.

Boolean Operators

The Boolean operators are shown in Table 6-2. The operands must have BOOLEAN data types and produce the BOOLEAN values TRUE and FALSE.

Table 6-2. Boolean Operators

Operator	Operation	Operand Type	Result Type
Dyadic			
AND	Logical AND	Boolean	Boolean
OR	Logical OR (inclusive)	Boolean	Boolean
Monadic			
NOT	Logical negation	Boolean	Boolean

The result of the operation

```
A AND B
```

is TRUE only if both A and B are TRUE. The result of

```
A OR B
```

is TRUE if either A or B is TRUE (inclusive OR). The result of

```
NOT A
```

is TRUE if A is FALSE, and *vice versa*.

Note that the exclusive OR operation is provided by the predeclared function XOR. That is, if a and b are

BOOLEAN variables or expressions, then XOR(a,b) is TRUE if a and b have different TRUE/FALSE values; otherwise, the result is FALSE.

Relational Operators

The relational operators are shown in Table 6-3. All take two operands and produce BOOLEAN results.

Table 6-3. Relational Operators

Operator	Operation	Operand Types
=	Equality	Ordinal types, floating-point types, system types, pointers, sets, and strings
<>	Inequality	Same as =
<	"Less than"	Ordinal types, floating-point types, and strings
>	"Greater than"	Same as <
<=	"Less than or equal to"; set inclusion	Ordinal types, floating-point types, sets, and strings
>=	"Greater than or equal to"; set inclusion	Same as <=
IN	Set membership	Left: ordinal type *T*; right: set whose base type is such a type

The operands of all operators except IN must be two expressions of compatible types, or two of the same set type, or else one must be INTEGER and the other REAL or DOUBLE.

The data types allowed as operands include the types listed in the table and user-defined types denoting these types. Note that arrays and records (except PACKED ARRAY OF CHAR used as a string) cannot be operands of relational operators.

Note that the system data types PROCESS, AREA, EVENT, SEMAPHORE, MESSAGE, PORT, NAME, and DEVICE can be compared only with the equality and inequality operators.

Equality (=) and Inequality (< >)

The result of an equality operation is TRUE if the two operands are equal; the result of the inequality operation is TRUE if they are not equal.

Two sets are equal if they contain exactly the same elements.

Two strings are equal if all their characters are identical. For strings of different lengths, the shorter string is implicitly extended with spaces. (Note that uppercase and lowercase letters are not identical.)

Two pointers are equal if they point to the same object or have the value NIL. One pointer can be of type ↑ ANYTYPE; otherwise, both must point to the same type. Equality and inequality are the only valid operations on pointers.

"Less Than," etc. (<, >, < =, > =)

The result of one of these operations is TRUE if the relation is true.

For example,

```
A < B
```

is TRUE if the value of A is less than B's. The relation of ordinal types depends on their ordinal values; with characters, this is the same relation as alphabetical order.

Sets can be compared only with < = and > =, not with < or >. If A and B are sets, the operation

```
A < = B
```

is TRUE if and only if all of A's elements are elements of B (that is, A is *included in* B, or A is a *subset of* B). Conversely,

```
A > = B
```

is TRUE only if B is included in A.

When these operators are used to compare strings, they denote *lexicographic relations*; that is, the rules by which words are ordered alphabetically. For example,

```
'thomas' < 'mary'
```

is FALSE, and

```
'1234' < '56834'
```

is TRUE.

Formally, the definition of S1 < S2 is as follows, where S1 and S2 are strings of equal length and, for example, S1[2] denotes the second character in the string S1:

```
S1 < S2 if and only if:
    there exists a p in the closed interval [1..n] such
    that, for all i in the interval [1..p – 1].
    S1[i] = S2[i] AND
    S1[p] < S2[p]
```

This might also be stated as, "S1 is less than S2 only if, at the first place they disagree, S1's letter is less than S2's."

Furthermore, strings of unequal length can be compared; effectively, the shorter string is extended with spaces before the comparison is made. Note that the ordinal value of the space character (32) is less than that of any letter or digit.

The following relations are TRUE in this implementation:

```
'THOMAS' < 'Thomas'
'THOM' < 'THOMAS'
'tom and mary' < 'tomandmary'
'tom_andmary' < tomandmary'
'TOM_ANDMARY' > 'TOMANDMARY'
```

Set Membership (IN)

The left-hand operand of IN must have an ordinal value of type *T*, and the right-hand operand must denote a SET OF *T*.

The result of the operation

```
A IN B
```

is TRUE if ordinal value A is an element of set B.

Set Operators

The set operators are shown in Table 6-4. Both operands must be sets with the same base ordinal type, and the result is a set of the same type.

Table 6-4. Set Operators

Operator	Operation	Operand Type
+	Set union	Compatible sets
−	Set difference	Compatible sets
*	Set inter-section	Compatible sets

The plus operator (+) produces the *union* of two sets. That is, it produces a new set containing all the members from each set. Logically, this is the *inclusive OR* operation, because an element appears in the resulting set if it is present in either operand, including the case in which it is present in both operands. For example,

```
cset := cset + ['B']
```

adds the letter B to the elements already in cset.

The minus operator (−) produces the *difference* between two sets. That is, it produces a new set containing those members present in the left-hand set but not present in the right-hand set. For example,

```
cset := ['a'..'d'] - ['c']
```

assigns the set ['a','b','d'] to cset.

Note that the difference operation depends on the order of the operands and so *does not* answer the question, "Which elements belong to only one set or the other?" This question is answered by using the predeclared function XOR, as described below.

The asterisk operator (*) produces the *intersection* of two sets. That is, it produces a new set containing the elements present in both. Logically, this is the *AND* operation, because an element appears in the resulting set only if it appears in both operands. For example,

```
cset : = ['a'..'e'] * ['c'..'f']
```

assigns the set ['c'..'e'] to cset.

In addition to the operators described above, the predeclared function XOR provides the logical *exclusive OR* operation on two sets. That is, an element appears in the resulting set if it appears in one set but *not* if it appears in both. XOR(a,b) has the same result as (a−b)+(b−a).

The following example shows more results of these operations:

```
VAR
   cset,alphabet: SET OF CHAR;
   uppercase: SET OF 'A'..'Z' : = ['A'..'Z'];
   lowercase: SET OF 'a'..'z' : = ['a'..'z'];
BEGIN
   alphabet : = uppercase + lowercase; { All the
   letters. }
   cset : = ['a','A','b','B'] * uppercase; { The set
   ['A','B']. }
   cset : = uppercase - ['A'..'W']; { The set
   ['X','Y','Z']. }
   cset : = XOR(['a','b'],['b','c']); {The set ['a','c']. }
```

Set Constructors

Set constructors are used in expressions and initializers to enumerate elements of the base type. The syntax is shown in Figure 6-5.

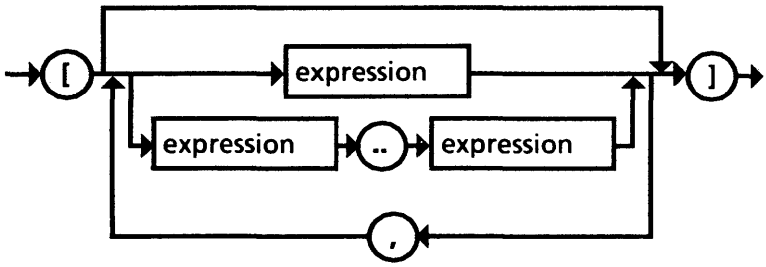


Figure 6-5. Set Constructor Syntax

Each expression must be of the same ordinal type (ignoring subrange limits), which is the base type of the constructed set. The form

```
expression1 .. expression2
```

denotes the set of all elements *e* of the base type such that expression1 ≤ *e* ≤ expression2.

If expression1 > expression2, then the form denotes the empty set of the base type, which can also be indicated by a set constructor containing no expressions ([]). For example:

```
TYPE
charset = SET OF CHAR;

VAR
alphabet: charset : = ['A'..'Z','a'..'z']; { The set of
letters. }
empty: charset : = ['z'..'a']; { The empty set of
characters. }
empty2: charset : = []; { Same thing. }
```

Here, the type of all three set constructors is SET OF CHAR.

Note: If any expressions in the constructor are non-constant, the amount of storage required to represent the constructed set may not be known at compile time. This can result in inefficient code, so the compiler tries to limit the required storage in various ways. For example, if the set's base type is reasonable (almost anything but INTEGER) or the constructor is the source expression in an assignment to a set variable of modest size, the code will be reasonable.

Concatenation Operator for Strings

The concatenation operator is the plus sign (+). Applied to two string expressions, the concatenation produces a new string value. For example:

```
VAR
   vstring : VARYING_STRING(15);
BEGIN
   vstring : = 'abcdef' + 'ghijkl';
```

Here, the result of the concatenation (and, thus, the value of vstring) is 'abcdefghijkl'.

Chapter 7
Pascal Statements

VAXELN Pascal provides several statements that control the actions performed in a program. These are:

- The assignment statement
- The null statement
- The compound statement
- The conditional statements CASE and IF
- The loop statements FOR, REPEAT, and WHILE
- The WITH statement
- The GOTO statement
- Procedure calls

In addition, all statements can be labeled, as possible targets of GOTO statements.

This chapter discusses the general statement syntax, including labels, followed by a discussion of each statement and its syntax. Procedure calls are discussed in Chapter 8, "Procedures and Functions."

General Statement Syntax

The general statement syntax for VAXELN Pascal statements is shown is Figure 7-1.

Labels

Labels mark locations in a program that are possible destinations for GOTO statements. The label is separated from the labeled statement by a colon (:).

A label can be a literal integer constant or any valid identifier, as shown in Figure 7-2.

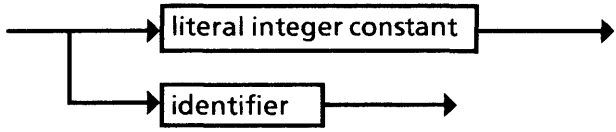


Figure 7-2. Label Syntax

If a statement is labeled by a literal integer constant, it must be an unsigned decimal integer. The compiler treats it as though it is the identifier whose spelling is given by the decimal digits, ignoring leading zeros.

If a statement is labeled by an identifier, the identifier is declared as a label, with its scope being the block containing the statement.

In either case, the integer or identifier must not occur as the label of another statement in the same block.

The following are examples of labels:

```
error_message: WRITELN('Could not continue. ');
10: WRITELN('Terminating driver process. ');
```

The corresponding GOTO statements are:

```
GOTO error_message;
GOTO 10;
```

Since labels are declared implicitly by their use, the declaration of labels in a routine body is not required, but is allowed for compatibility with standard Pascal. The label declaration syntax is shown in Figure 7-3.

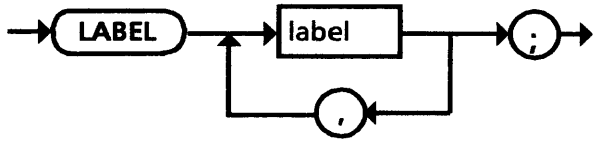


Figure 7-3. Label Declaration Syntax

Note: If any label in a block is declared with a label declaration, all labels in the block must be declared, and all declared labels must be used as statement labels in the same block.

Assignment Statement

The syntax for the assignment statement is shown in Figure 7-4.

Figure 7-4. Assignment Statement Syntax

Execution of the assignment statement evaluates the target variable reference on the left-hand side and the source expression on the right-hand side and assigns

the value of the expression to the data item denoted by the target reference.

The target and source data types must be assignment compatible, as explained below. The target reference must not denote a data item that is invalid as an assignment target. In particular:

- A variable declared in a VAR declaration with the READONLY or VALUE attribute.
- A value parameter with the READONLY or LIST attribute.
- A conformant extent or value parameter used as an explicit extent in the same routine heading. (See Chapter 8, "Procedures and Functions," for more information on conformant parameters.)
- The control variable of a FOR loop, when the assignment is inside the loop or in a subblock of the block containing the FOR statement.
- A variable that is a file or contains a file.

These rules apply to the variable used on the left-hand side of an assignment statement, a variable passed as an argument to a predeclared routine that modifies the argument, and a variable passed as an argument to a VAR parameter of a routine that modifies the parameter. Performing an assignment to such a variable by another name (such as a VAR parameter or via pointers) is an unpredictable error.

Assignment Compatibility

In any situation where a source value is assigned to a target variable, Pascal requires that the source value's type be assignment compatible with the target variable's type. The principal contexts in which this rule applies are assignment by an assignment

statement, as described above, and passing an argument value to a value parameter.

The general rules for assignment compatibility are given in Table 7-1. The following notes apply:

1. Regarding subranges of ordinals.

   The type of an ordinal value is always a basic ordinal type, not a subrange type. For example:

   ```
   VAR k: 0..255; n: INTEGER;
   n := k;
   ```

   Here, the value of k is of type INTEGER and is assignment compatible with k. If n was also of subrange type (for example, -128..127), the value of k would be assignment compatible, but a range violation would occur if the value fell outside n's subrange.

2. Regarding INTEGER.

   A REAL or DOUBLE source value is not considered assignment compatible with INTEGER, because the assignment can reasonably be done with either rounding or truncation. Use the ROUND or TRUNC functions (see Chapter 9, "VAXELN Routines") to specify the desired conversion.

3. Regarding BYTE_DATA.

   If a value parameter has type BYTE_DATA(*n*), any argument value is treated as assignment compatible. It is converted to BYTE_DATA(*n*) as described under the CONVERT function (see Chapter 9). Note that this extra freedom does not apply in other contexts.

4. Regarding sets.

   Assignment compatibility for sets requires only that the source set value and target set type have the same element type. A range violation occurs in either of the following cases:

   - The source value contains an element such that ORD(element) is less than the minimal element of the target type.
   - The source expression contains an element such that ORD(element) is greater than the maximal element of the target AND is less than or equal to the index of the highest bit in the target variable's representation (bits beyond that are simply discarded).

   Note that truncating nonzero source bits in the copy is not considered an error. Indeed, when the source is an expression, the compiler may avoid computing bits outside the range required by the target's size. For example:

```
VAR
  s : SET OF 0..15; { One longword }
  r : PACKED RECORD
    f: PACKED SET OF 5..15; { 16 bits. }
    END;
  .
  .
s : = [0..31];    {Range violation, 16..31 not in
                   range.}
s : = [32..63];   { OK; assigns empty set. }
r.f : = [0..15];  { Range violation; 0..4 not in
                   range. }
r.f : = [5..31];  { OK; assigns [5..15]. }
```

Null Statement

The syntax for the null statement is shown in Figure 7-5.

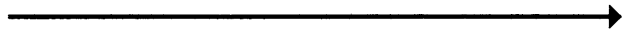


Figure 7-5. Null Statement Syntax

Execution of the null statement has no effect. The only real purpose of the statement is to provide an explicit null alternative in an IF statement.

Compound Statement

The syntax for the compound statement is shown in Figure 7-6.

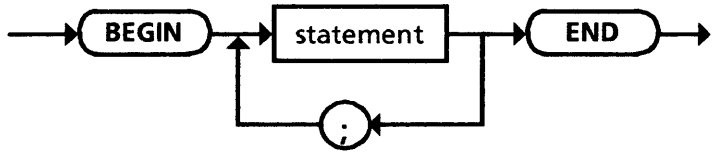


Figure 7-6. Compound Statement Syntax

A compound statement groups a sequence of statements together as a single larger statement so they can be used as the body of a routine, an alternative in an IF statement, or the body of a WHILE statement, for example.

Note that the statement sequence needs semicolons only to separate statements, not to terminate them. That is, a sequence of n statements requires only $n - 1$ semicolon separators. Extra semicolons have no effect on the program's meaning and generate no code.

Executing the compound statement performs the contained statements in order, unless the order is modified by execution of a GOTO statement.

CASE Statement

The syntax for the CASE statement is shown in Figure 7-7.

All of the limited ordinal constants occurring as *case constants* must have the same data type, and the expression must also have this type. (The syntax for limited ordinal constants is given in Chapter 4, "Constants.") The range of values in the case constants must not exceed 32,767.

When a CASE statement is executed, the expression is evaluated. If its value is equal to one of the case constants, the corresponding statement is executed. If the value of the expression is not equal to one of the case constants, the first statement in the OTHERWISE clause is executed. If the OTHERWISE clause is omitted, and the value of the expression is not equal to one of the constants, a range violation occurs.

For example,

```
CASE ch OF
   'A': procedure_A;
   'b': procedure_b;
   OTHERWISE IF ch IN alphabet THEN...
     ELSE...;
END;
```

calls (selects) procedure_A if the expression ch equals the uppercase letter A. It selects procedure_b if the expression equals b. The IF statement is executed if the expression equals neither of the constants. Only one case (here, either 'A', 'b', or OTHERWISE) is selected. You can define as many possible actions as there are possible values of the expression, and, with the OTHERWISE clause, you can define an action to take if no other case is selected.

In most cases, the compiler generates a VAX CASE instruction for a CASE statement. This requires a word of storage for each value in the range of constants

(minimum to maximum), which can be very inefficient for a sparse range. For example, the statement

```
CASE integer_expression OF
   1: procedure_a;
   30000: procedure_b;
   OTHERWISE procedure_c;
END;
```

allocates 30,000 words for possible values of integer_expression, even though only two of these values actually appear as case constants. The guideline, then, is to avoid situations where the case constants specify a very large range but only a very few values in the range are tested; IF is much more efficient in such cases. The type CHAR and most enumerated types have ranges small enough that this matter is usually irrelevant.

CASE is often useful in examining the result of a WAIT_ANY procedure call, to determine how the wait was satisfied and take appropriate action. For example:

```
.
.
VAR
   ok: EVENT;
   unit_available: SEMAPHORE;
   other_process: PROCESS;
   satisfier: INTEGER;
.
.
WAIT_ANY(
   ok,
   unit_available,
   other_process,
   RESULT : = satisfier,
   TIME : = TIME_VALUE('0 00:00:01.0')
   );
```

```
CASE satisfier OF
   0: { Timeout (one second elapsed)...} ;
   1: { Event ok was signaled... } ;
   2: { Unit is available... } ;
   3: { Other process terminated...} ;
   .
   .
   END;
END;
```

Here, WAIT-ANY returns an integer (satisfier) identifying the argument that satisfied the wait. The possible return values are all within a small range (the number of arguments present in the call), and you usually cannot predict the chances of any particular value being returned. For both these reasons, and for readability, CASE is a good choice for processing the result of the procedure call.

IF Statement

The syntax for the IF statement is shown in Figure 7-8.

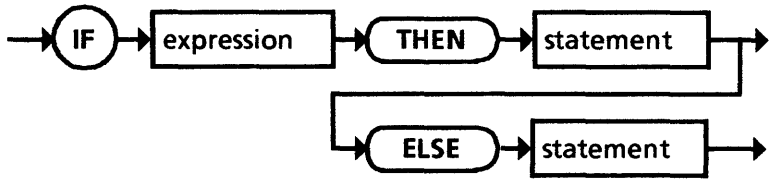


Figure 7-8. IF Statement Syntax

Execution of the IF statement evaluates the expression, whose type must be BOOLEAN. If the resulting value is TRUE, the statement following THEN is executed. If

the value is FALSE and an ELSE clause is present, the statement following ELSE is executed.

In a series of THEN ... ELSE clauses, each ELSE always matches the nearest unmatched THEN. For example:

```
IF A = 1
THEN IF A = 2
        THEN procedure1
        ELSE procedure2;
```

Here, the clause ELSE procedure2 matches the clause THEN procedure1.

Note that the null statement can be used to provide an explicit null alternative in an IF statement.

FOR Statement

The syntax for the FOR statement is shown in Figure 7-9.

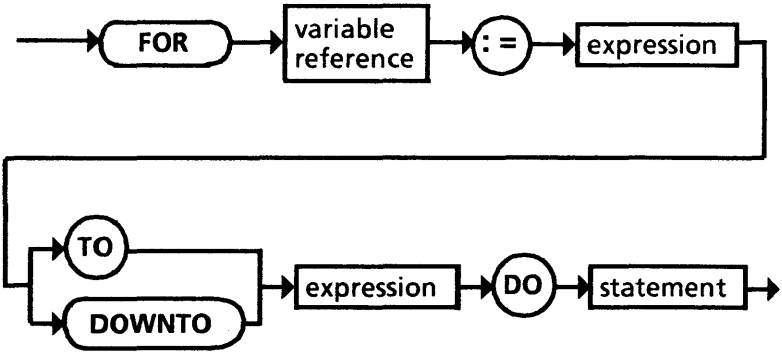


Figure 7-9. FOR Statement Syntax

The FOR statement executes a body of statements repeatedly, while it assigns a sequence of values to a variable reference called a *control variable*. The body is executed a fixed number of times, possibly zero.

The control variable must be an entire variable of ordinal type. It must be declared in the same block as the FOR statement. Within the statement itself or any subblocks of the FOR statement's block, the variable must not be the target of an assignment, READ call, or WRITE_REGISTER call, passed to a VAR parameter, or used in any other operation that might change its value; this is to guarantee that the FOR loop always executes the specified number of times. The reserved word TO indicates that the variable is incremented on each repetition. DOWNTO indicates that the variable is decremented on each repetition.

Note that the value of the control variable is undefined after the last repetition of the loop, unless you leave the loop with a GOTO statement. You should not write programs that depend on the final value of the control variable after the loop terminates normally. Taking the address of the variable (with the ADDRESS function) is illegal.

The type of the expressions must be compatible with the control variable. The first expression supplies the value of the variable for the first execution of the FOR loop, if the loop is executed at all. (With TO, this expression must be less than MAXINT; overflow is possible.) The second expression supplies the value used to control the termination of the FOR loop.

The body of a FOR statement (the statement following the reserved word DO) can be any statement, including compound statements and FOR statements. The statement can be (or contain) a GOTO statement that transfers control to a label outside the FOR loop.

If the loop is exited by a GOTO statement, the control variable has the last value assigned to it by FOR, and its control variable is available for use outside the loop. This feature is sometimes useful for responding to an abnormal condition, such as invalid input.

Use the FOR statement when you know beforehand how many times an operation must be repeated. For example, the following program converts six-digit binary integers to decimal:

```
PROGRAM convert(INPUT,OUTPUT);
{ Convert binary numbers to decimal. }

LABEL 10,20;

CONST digits = 6;

TYPE numarray = ARRAY[1..digits] OF INTEGER;

VAR
chararray : PACKED ARRAY[1..digits] OF CHAR;
unparray  : ARRAY [1..digits] OF CHAR;
oldnumber : numarray;     i: INTEGER;

FUNCTION decimal(old: numarray) : INTEGER;
   VAR i,newnumber: INTEGER;
   BEGIN
     newnumber : = old[1];
     FOR i : = 2 TO digits DO
       newnumber : = old[i] + newnumber * 2;
     decimal : = newnumber;
     END;

BEGIN

   READLN(chararray);
   UNPACK(chararray,unparray,1);
   FOR i : = 1 TO digits DO
     CASE unparray[i] OF
       '0': oldnumber[i] : = 0;
```

```
        '1': oldnumber[i] : = 1;
        OTHERWISE GOTO 10;
      END;
    WRITELN(decimal(oldnumber));
    GOTO 20;
10: WRITELN('The character ',unparray[i],
' is invalid; terminating.');
20: ;
END .
```

REPEAT Statement

The syntax for the REPEAT statement is shown in Figure 7-10.

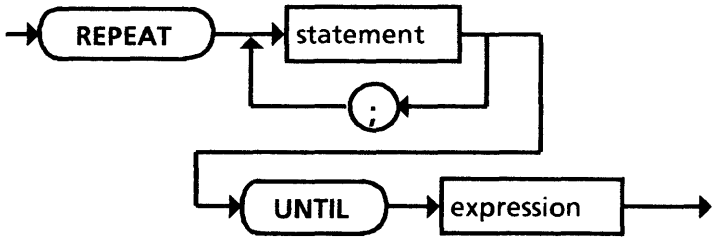


Figure 7-10. REPEAT Statement Syntax

The REPEAT statement executes a body of statements repeatedly, until a stated condition is true. The expression must be of type BOOLEAN. After each repetition of the statement (that is, "at the bottom of the loop"), the expression is evaluated. The REPEAT loop terminates when the expression is TRUE. Note that the body is always executed at least once.

Any statement is valid in the body of a REPEAT loop. A sequence of statements is delimited by the words REPEAT and UNTIL and need not be enclosed in BEGIN and END.

Use the REPEAT statement when you do not know how many times the loop should execute, but you know it should execute at least once; usually, it is necessary when the value of the expression is derived from the operation of the loop itself.

WHILE Statement

The syntax for the WHILE statement is shown in Figure 7-11.

Figure 7-11. WHILE Statement Syntax

The WHILE statement executes a body of statements repeatedly, as long as a stated expression is TRUE. The expression must be of type BOOLEAN. Before each repetition of the loop (that is, "at the top of the loop"), the expression is evaluated, and the body (the statement following the reserved word DO) is executed only if the result is TRUE.

Any executable statement is valid in the body of a WHILE statement, including another WHILE statement or a compound statement.

Use the WHILE statement when you do not know beforehand how many times the loop should execute,

and when it should not be executed at all unless a given condition is TRUE. Like REPEAT, WHILE is usually used when the condition for terminating the loop results from the action of the loop itself.

For example, the following program reads characters from INPUT until a nonalphabetic character is encountered or until end-of-file is encountered:

```
PROGRAM readchar(INPUT,OUTPUT);
{ Read characters from INPUT. }

CONST maxlength = 26;

VAR
   alphabet : SET OF CHAR;
   ch: CHAR;
   word: VARYING_STRING(maxlength);

BEGIN
   alphabet : = ['a'..'z','A'..'Z'];
   word : = '';
   IF NOT EOF THEN READ(ch);
   WHILE NOT EOF AND (ch IN alphabet) DO
     BEGIN
     word : = word + ch;
     READ(ch);
     END;

   WRITELN('Word: ',word)
END .
```

This program constructs a word from the characters it reads and writes the word to OUTPUT. It considers a word to be a series of the characters a–z and A–Z, terminated by a nonalphabetic character (including a space) or by the end of the file INPUT. To be general in purpose, the program must account for the very first character being nonalphabetic; that is, it cannot predict the first character. Therefore, the WHILE statement is

a better choice than REPEAT, which would have to make an assumption about the first character.

Similarly, words have a maximum length, but the precise number of characters in a word is not known beforehand; instead, the termination of the loop is controlled by the operation of the loop itself—reading a particular character. Therefore, FOR is not a good choice, since it would require all input words to have the same length.

WITH Statement

The syntax for the WITH statement is shown in Figure 7-12.

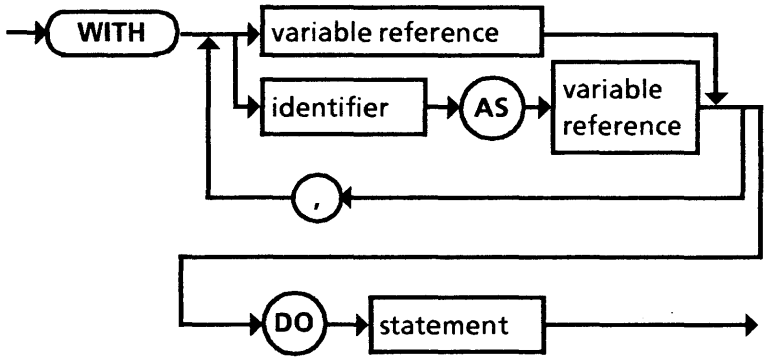


Figure 7-12. WITH Statement Syntax

Execution of the WITH statement establishes part of a program in which abbreviated references can be made to the fields of a record or in which an identifier is established as the name of the data item denoted by a variable reference.

The first variable reference in the syntax is a reference to a declared record. The variable is accessed before the body (the statement following the reserved word DO) is executed, and a reference is established for the duration of that statement.

The variable reference following the reserved word AS introduces a temporary variable that can be referenced within the body of the WITH statement (a typecast variable is often used here, as shown in the second example below).

Note: When a WITH statement has multiple references, they are evaluated in order, and each may apply to the following ones.

The statement in the body of a WITH statement is usually a compound statement. For the extent of this statement, you can refer to fields in the specified record or records by using their field names only: within the scope of the WITH statement, fieldname means "ref.fieldname," where ref is the established reference.

For example:

```
TYPE rtype = RECORD
   person : VARYING_STRING(80);
   stats : RECORD
     salary : REAL;
     hiredate : LARGE_INTEGER;
     END { stats. }
   END; { rtype. }
VAR
   r : rtype;
   rptr : ↑rtype;
   rarray : ARRAY [1..10] OF rtype;
BEGIN
   WITH r, r.stats DO BEGIN
     person : = 'Arturo Toscanini';
```

```
    salary : = 15000.00;
    hiredate : = TIME_VALUE(
      '1-JAN-1939 00:00');
  END;
  NEW(rptr);
  rptr↑ : = r;
  WITH rarray[1].stats DO
  hiredate : = rptr↑.stats.hiredate;
END;
```

Here, the WITH statements establish references to the records r, r.stats, and rarray[1].stats. Within the statements' bodies, these references apply to field names that are not preceded by a record reference.

If a temporary variable is introduced by the AS clause of the WITH statement, you can refer to it within the statement. The WITH ... AS construction is especially useful with typecasting. In the following example, the name str is introduced to represent a record typecast to type VARYING_STRING:

```
CONST k = 100;
TYPE { Model of type VARYING_STRING. }
  vstring(n : 0..32767) = PACKED RECORD
    length : [WORD] 0..32767;
    body : STRING(n);
  END;
VAR
  x : vstring(k);
BEGIN
  .
  .
  WITH x, str AS x::VARYING_STRING(k) DO
  BEGIN
  { Here, you can refer to x's fields (the string's
  length and body) by the simple names 'length' and
  'body' or to the name str, which is x treated as a
```

```
    VARYING_STRING. Without the AS clause, you
    could refer only to the fields 'length' and 'body.' }
    END; { End of WITH. }
END;
```

GOTO Statement

The syntax for the GOTO statement is shown in Figure 7-13.

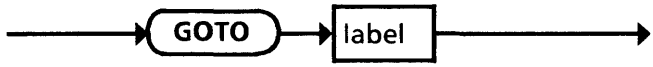


Figure 7-13. GOTO Statement Syntax

Execution of the GOTO statement unconditionally transfers control (or "branches") to a specified label.

The label is a literal integer constant or an identifier, as explained under "Labels," earlier in this chapter. Either must occur as a statement label for a statement in the block containing the GOTO statement or in some higher-level block containing that block (this is referred to as an *up-level GOTO*).

Restrictions

The following restrictions apply to every use of the GOTO statement.

Branch into Structured Statement. A GOTO statement must not transfer control from the outside to the inside of a structured or compound statement.

For example:

```
.
.
GOTO case_stmt;                          { Invalid. }
CASE letter OF
   'A': case_stmt: BEGIN
   WRITELN('a'); GOTO case_stmt          { Valid. }
   END;
   'B': WRITELN('B');
   OTHERWISE ...;
END;
```

Up-level GOTO. An up-level GOTO is one that specifies a label in a higher-level block containing the block to which the GOTO statement belongs. In this case, the target label must label a statement immediately contained in the higher-level block. (That is, it must label a statement in that block's main sequence of statements). The compiler detects all violations of this rule.

For example:

```
PROCEDURE high;
   PROCEDURE low;
     BEGIN                  { Low code. }
       GOTO mainlabel;      { Valid up-level. }
       GOTO innerlabel;     { Invalid up-level. }
     END;                   { Low code. }
   BEGIN                    { High code. }
     low;
     .
     .
```

```
FOR I : = 1 TO 100 DO BEGIN
  .
  .
  innerlabel: ...   { Statement. }
  END; { FOR }
mainlabel: EXIT;  { Labels a statement in the
                    block's main BEGIN-END. }
END;                { High code. }
```

An up-level GOTO terminates the block activation that executes it and all activations between that block and the target block activation. This feature is useful in "unwinding" the call stack from an exception handler. Call frames are removed from the stack, beginning with the GOTO's (exception handler's) frame, until the target block is reached. (See Chapter 13, "Errors and Exception Handling," for more information.)

Chapter 8
Procedures and Functions

Introduction

Procedures and functions give you a means of putting a frequently used computation in a single package, which can then be invoked from several points in a program.

This chapter discusses the following topics:

- Procedure and function declarations (including procedure and function headings, parameter lists, function results, procedure and function types, directives, and in-line routines)
- Procedure and function calls (including arguments and argument lists)
- Parameters and argument passing (including the parameter/argument relationships for VAR and value parameters, procedural parameters, conformant parameters, OPTIONAL parameters, and LIST parameters)
- Calling conventions

The meanings of *procedure* and *function* are quite similar, and, in this manual, both are meant by the term *routine*. When invoked, or called, a routine performs the actions you have defined in its declaration.

A function also takes on a value, called the *function result*. To invoke a finction, its name (possibly with a list of arguments) is simply used in the program as if it were an expression of a particular type.

Procedure and Function Declarations

Procedure and function declarations have almost the same form. The only differences are the reserved word (PROCEDURE or FUNCTION) in the heading, and the fact that a function heading specifies a result type.

The syntax of procedure and function declarations is shown in Figures 8-1 and 8-2, respectively.

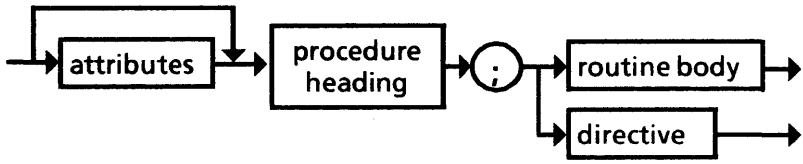


Figure 8-1. Procedure Declaration Syntax

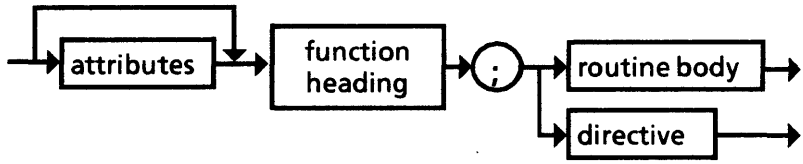


Figure 8-2. Function Declaration Syntax

A procedure or function declaration declares the procedure or function identifier in its heading as the name of a routine or (if the PROCEDURE_TYPE or FUNCTION_TYPE directive is used) as the name of a procedure or function type. The heading declares the

routine's parameters (if any) and, for a function, its result type. (See "Procedure and Function Headings," later in this section.)

The routine body, if present, contains the local declarations and code for the procedure or function. Routine bodies have the same form for procedures, functions, PROGRAM blocks, process blocks, and interrupt service routines. (See Chapter 2, "Program Structure," for the syntax of routine bodies.)

The syntax of directives is shown in Figure 8-3.

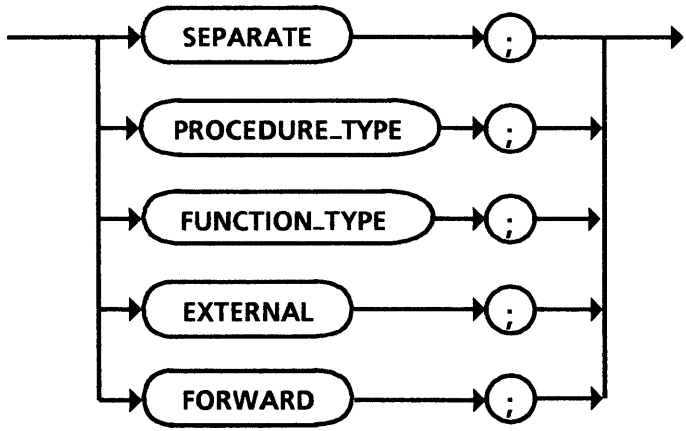


Figure 8-3. Directive Syntax

The directives SEPARATE and FORWARD indicate that the text of the routine body is elsewhere. The EXTERNAL directive means that the routine body is defined in another programming language. SEPARATE and EXTERNAL are only allowed at the outer level.

Note that the scope of parameter declarations in the procedure or function heading includes the routine body even in the cases in which its text is elsewhere (that is, when the SEPARATE or FORWARD directive is present).

If a procedure or function declaration contains a routine body, one of the mutually exclusive attributes UNDERFLOW, NOUNDERFLOW, or INLINE may be specified. UNDERFLOW and NOUNDERFLOW apply to the code of the routine body (see Chapter 2 for more information). INLINE specifies that each call to the routine is to be expanded into code inline (see "INLINE Procedures and Functions," later in this section.)

Procedure and Function Headings

A heading can occur in a procedure or function declaration or in a parameter list. The syntax of procedure and function headings is shown in Figures 8-4 and 8-5, respectively.

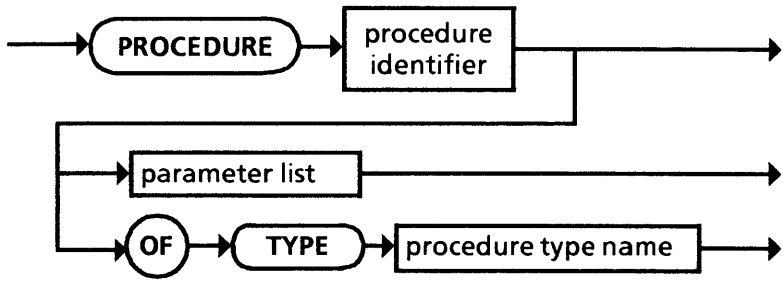


Figure 8-4. Procedure Heading Syntax

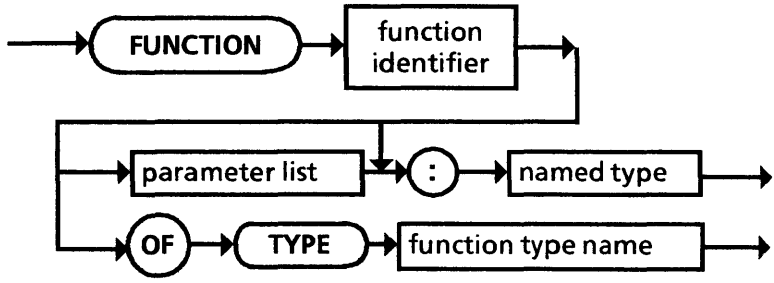


Figure 8-5. Function Heading Syntax

The procedure or function identifier is declared as:

- The name of a procedure or function if the heading occurs in a procedure or function declaration without the directives PROCEDURE_TYPE or FUNCTION_TYPE.
- The name of a procedure or function type if the heading occurs in a procedure or function declaration and is followed by the directive PROCEDURE_TYPE or FUNCTION_TYPE.
- The name of a procedural parameter if the heading occurs in a parameter list.

In the first two cases, the scope of the declaration is the block containing the procedure or function declarations. The third case is governed by the rules for parameter declarations (see "Parameter Lists," later in this section).

The heading's parameter list, if present, declares the routine's parameters. The named type following the colon (:) in a function heading specifies the function's result type. Note that for name interpretation, the

function's result type is within the scope of parameter declarations in the parameter list.

Unless the construction OF TYPE is used, a function heading must give a result type, and absence of a parameter list in a procedure or function heading means that the routine has no parameters.

The construction OF TYPE followed by a procedure or function type name is used as an alternative to specifying an explicit parameter list and function result type. Instead, these are taken from the named procedure or function type. (For more information, see "Procedure and Function Types," later in this section.)

Parameter Lists

A parameter list declares the parameters of a procedure, function, process block, or interrupt service routine. The syntax of parameter lists is shown in Figure 8-6.

The scope of a parameter declaration is:

- The parameter list in which it occurs; plus
- The result type of the function, if it's a parameter of a function; plus
- The routine body, if the parameter list is in a routine declaration that has a body, whether or not its text is elsewhere.

A parameter is either a *value parameter*, a *VAR parameter*, or a *procedural parameter*. The latter is declared by the occurrence of an entire procedure or function heading in the parameter list.

Normally, a routine call specifies an argument to be passed to each parameter of the called routine, but some flexibility may be obtained by use of default values (that is, initializers) for value parameters, the OPTIONAL attribute for VAR and procedural parameters, and the LIST attribute.

The section "Procedure and Function Calls," later in this chapter, explains how the arguments in a call are associated with particular parameters of a called routine. The section "Parameters and Argument Passing" describes in detail the relationship between each of the various types of parameters and its argument.

The remainder of this subsection gives some general rules for the contents of a parameter list. The rules given here apply to all routines, but there are additional restrictions on the parameters of process blocks and interrupt service routines.

Data Type for a Parameter

A named type supplies the parameter's data type. The named type can be a type name, a bound flexible type, or a pointer to either of these (for example, ↑STRING(10)). The bound flexible type may refer to other value parameter names in extent expressions and it may define a conformant extent using the angle bracket notation (for example, STRING(<n>)). See "Conformant Parameters," later in this chapter, for more information.

An ISO conformant type may be used instead of a named type to define a conformant array parameter.

See "ISO Conformant Extents," later in this chapter, for more information on conformant array parameters.

Attributes of Parameters

The attributes OPTIONAL and LIST apply specifically (and only) to parameters; they are explained in separate sections, later in this chapter. READONLY applies to value parameters, it is covered in the section on value parameters. REFERENCE applies to certain value parameters; it pertains to the calling conventions and is covered in that section.

Apart from the preceding, the only attributes allowed are BIT, BYTE, WORD, and LONG, which may be applied to ordinal types and small set types (that is, where ORD(maxelement) < 32), as explained in Chapter 3, "Data Types."

Default Values for Value Parameters

The type of a value parameter may be followed by the construction ":= initializer". The initializer specifies a constant default value for the parameter. This value is passed to the parameter if a routine call does not specify an argument for it. The initializer's form depends on the type of the parameter, as explained in Chapter 4, "Constants." The parameter must be constant sized.

Function Result

A function call occurs as a factor in an expression. Like any other Pascal expression, it has a data type. This is specified by the function's result type, which follows the colon (:) in the function heading. For example,

```
FUNCTION f(x: REAL) : ↑REAL;
```

defines a function of type ↑REAL.

Like the type of a parameter, the result type of a function is specified as a named type. This may be a bound flexible type, depending on the function's value parameters or conformant extents.

For example:

```
FUNCTION reverse(s: [READONLY] STRING(<n>)) :
   STRING(n);
```

Note that the result type cannot be a file type nor contain a file type.

Within a function's routine body, the function name is used in a restricted way as the name of a local variable, called the *result variable*. The function name only has this interpretation when it occurs as the variable name in a variable reference on the left-hand side (that is, the target side) of an assignment statement.

At normal termination, the value of the result variable becomes the value of the function call. It is an unpredictable error if the result variable does not have a value or if its value is incomplete (for example, only part of an array result is defined) or inconsistent with the result type (because of typecasting).

For example:

```
FUNCTION reverse(s: [READONLY] STRING(<n>)) :
   STRING(n);
VAR k: INTEGER
BEGIN
   FOR k := 1 TO n DO
      SUBSTR(reverse,k,1) := SUBSTR(s,n-k + 1,1);
END;
```

Here, it would be an error if the loop was from 2 TO n, because the first character of the result would be undefined.

SEPARATE Procedure and Function Declarations and Separate Routine Bodies

An outer-level procedure or function declaration may contain the SEPARATE directive instead of a routine body. This means that the routine's actual body is given elsewhere. Normally, the separate body is defined in another module, but it may be in the same module containing the SEPARATE declaration.

The form of a separate routine body is shown in Figure 8-7.

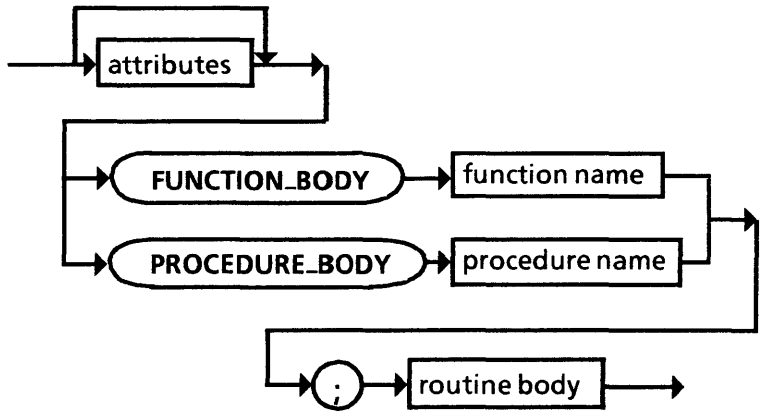


Figure 8-7. Separate Routine Body Syntax

You can apply one of the attributes UNDERFLOW and NOUNDERFLOW to the code of the routine body, as explained in Chapter 2, "Program Structure."

The following rules govern the use of the SEPARATE:

- SEPARATE and separate routine bodies are only allowed at the outer level.

- The procedure or function name in a separate routine body must be declared by a SEPARATE declaration, either in the current module or by an exported declaration from another module included in the current compilation.

One typical use of SEPARATE declarations is in gathering a group of related declarations together in a module, while keeping the routine bodies in distinct modules, so they can be edited without editing the declaration module.

Another use, analogous to the use of FORWARD in standard Pascal, is with mutually dependent modules. For example:

```
MODULE m1;
PROCEDURE p(i: INTEGER); SEPARATE;
PROCEDURE q;
   BEGIN
   .
   .
   p(2); { Call p. }
   END;
END; { End of module. }

MODULE m2;
PROCEDURE-BODY p;
   BEGIN
   .
   .
   q; { Call q; }
   END;
END; { End of module. }
```

Here, module m1 depends on a declaration in m2 (the procedure p). The SEPARATE directive allows the two modules to be compiled in the order m1, m2. Module m1 contains all the information needed to compile a call to

procedure p even though the procedure is defined elsewhere (and may not yet be written).

EXTERNAL Procedure and Function Declarations

An outer-level procedure or function declaration may contain the EXTERNAL directive instead of a routine body. This means that the routine's code is written in another programming language, such as C or MACRO. EXTERNAL is only allowed at the outer level.

Note that EXTERNAL can be used with the same meaning in place of the routine body of a process block declaration or interrupt service routine declaration. EXTERNAL can also be used as an attribute in outer-level variable declarations.

FORWARD Procedure and Function Declarations

A procedure or function declaration may contain the FORWARD directive instead of a routine body. This means that the routine's actual body occurs later (textually) in the block containing the FORWARD declaration. FORWARD is included only to support standard Pascal. It is not needed in VAXELN Pascal, because the declarations in a block may occur in any order.

The actual routine body for a FORWARD declaration appears as a procedure or function declaration of the same name, with a routine body, and without a parameter list or result type (for functions). Note that this is not shown in the syntax diagrams.

Procedure and Function Types

A procedure or function declaration may contain the PROCEDURE_TYPE or FUNCTION_TYPE directive instead of a routine body. This means that the

declaration declares the name of a procedure type or function type, rather than the name of a routine.

Any procedure heading may contain the construction "OF TYPE procedure type name" instead of a parameter list. This means that the procedure's parameters have the same declarations (names and types) as the parameters declared in the procedure type.

Similarly, any function heading may contain the construction "OF TYPE function type name" instead of a parameter list and result type. The parameter declarations and result type are taken from the function type. For example:

```
FUNCTION ftype(n,m: INTEGER;
    VAR arr: ARRAY[1..10] OF INTEGER;) : BOOLEAN;
    FUNCTION_TYPE;

FUNCTION f OF TYPE ftype;
    BEGIN ... END; { Body of function f. }
```

Here, the function f has the value parameters n and m, the VAR parameter arr, and the result type BOOLEAN.

Defining a procedure or function type is generally appropriate when working with procedural parameters. Note that the Data Access Protocol interface (module $DAP) makes extensive use of procedure and function types. In addition, the predeclared function type EXCEPTION_HANDLER is discussed in Chapter 13, "Errors and Exception Handling."

INLINE Procedures and Functions

The INLINE attribute may be specified in a procedure or function declaration containing a routine body. This means that for each occurrence of a routine call to the procedure or function, the routine's body will be expanded into code at the point of the call.

For example, if the following INLINE procedure declaration is specified:

```
[INLINE] PROCEDURE increment(VAR i: INTEGER);
   BEGIN i : = i + 1  END;
```

A call to the declared procedure, such as

```
VAR n: INTEGER;
.
.
increment(n);
```

results in generated code similar to the following:

```
INCL R3
```

An in-line routine has no VAX argument list or stack frame, and the VAX CALL instruction is not used. INLINE has no other effect on the meaning of a routine, but its use is subject to the restrictions described below, to ensure that the compiler can perform the in-line expansion.

The use of INLINE will generally decrease program execution time. This occurs both because of the elimination of the general procedure call overhead, and because the compiler's general optimization methods apply across the expanded code. INLINE is especially appropriate for small procedures and functions, but its use on large ones can pay off as well. A good practice is to try it and compare execution times or simply inspect the resulting code.

The use of INLINE can cause difficulties in debugging. There is no stack frame for an in-line routine and no debugger symbol table information for the expanded routine (variables or statement locations). The NOINLINE qualifier can be used on the EPASCAL command, to force in-line routines to be compiled as normal routines, with stack frames and with debugger

symbol table information. This qualifier applies only to in-line routines declared in the compilation unit, not to those defined in other modules.

Restrictions on INLINE Procedures and Functions

The following restrictions on in-line procedures and functions must be observed:

- The body of an in-line routine must not contain other routines.
- An in-line routine must not establish an exception handler, nor be used as an exception handler.
- An in-line routine must not declare file variables or variables with the READONLY attribute. (File parameters are permissible.)
- If an in-line routine is exported and it refers to an outer-level variable or routine *X*, then *X* must be an exported name.
- A parameter of an in-line routine must not have the LIST attribute.
- An in-line routine *R* must not call itself, either directly or by calling another in-line routine that calls *R*. (However, it is permissible to call a non-in-line routine that calls *R*.)
- INLINE applies only to a complete procedure or function declaration; it must not be applied to a procedural parameter or to a SEPARATE, EXTERNAL, FORWARD, PROCEDURE_TYPE, FUNCTION_TYPE, PROCEDURE_BODY, or FUNCTION_BODY declaration. It cannot be used with the OF TYPE construction or with the UNDERFLOW or NOUNDERFLOW attribute.
- In-line routines must not be passed as arguments.

- Predeclared names used within the executable (BEGIN ... END) part of an exported routine should not be redeclared at the outer level of any module using the exported in-line routine; otherwise, when the compiler expands the in-line routine, it will misinterpret the predeclared name with unpredictable effects.

Procedure and Function Calls

Procedure and function calls have the same syntax and the same interpretation, except as follows: a procedure call occurs as a statement and does not return a result; a function call occurs as a factor of an expression and returns a result value.

When a routine call is executed, its arguments (if any) are associated with the routine's parameters and the routine's body is then activated. If the routine body terminates normally, control returns to the point of invocation and normal statement sequencing or expression evaluation continues. (See Chapter 2, "Program Structure," for more information on routine body activation and termination.)

The syntax of procedure and function calls is shown in Figures 8-8 and 8-9, respectively.

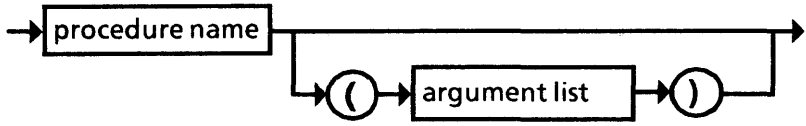


Figure 8-8 Procedure Call Syntax

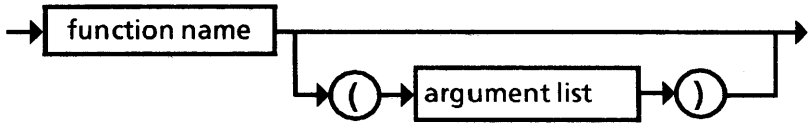


Figure 8-9 Function Call Syntax

The procedure or function name is the name of the routine to be called. This name is usually one declared by a procedure or function declaration. However, it can be the name of a procedural parameter, or it can be a call to the ARGUMENT function, accessing a particular argument routine passed to a procedural parameter with the LIST attribute.

The argument list in a procedure or function call specifies arguments to be passed to some or all of the called routine's parameters. As explained in the following subsection, either positional or nonpositional notation can be used to associate an argument with its corresponding parameter.

Once the parameter corresponding to a given argument is determined, the argument is interpreted in accordance with the particular properties of the parameter. This process is explained fully in the section "Parameters and Argument Passing," later in this chapter. Briefly:

- If the parameter is a value parameter, the argument is interpreted as an expression.
- If the parameter is a VAR parameter, the argument is interpreted as a variable reference.

- If the parameter is a procedural parameter, the argument is interpreted as a routine name (like the routine name in a procedure or function call).

Argument Lists

The argument list in a procedure or function call associates arguments with some or all of the called routine's parameters. An argument must be specified for a parameter unless it is a value parameter declared with a default value, an OPTIONAL VAR or procedural parameter, or a parameter with the LIST attribute. If no argument is given for a value parameter with a default value, the compiler supplies the default value as the argument. (The treatment of OPTIONAL and LIST parameters is explained in later sections of this chapter.)

The syntax for argument lists and arguments is shown in Figures 8-10 and 8-11, respectively.

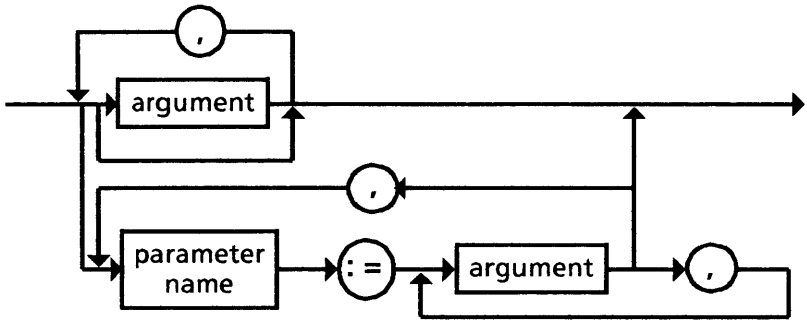


Figure 8-10 Argument List Syntax

Note: In the above syntax, a completely empty argument list is not allowed. The maximum number of arguments allowed is 253.

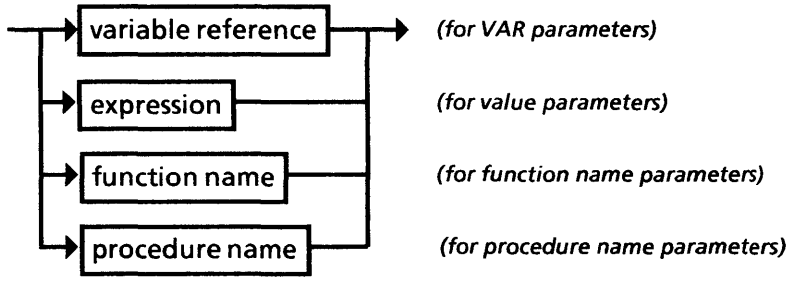


Figure 8-11 Argument Syntax

There are three possibilities for an entry in an argument list:

- Positional argument. This is an argument not preceded by "parameter name :=". Positional arguments are associated in left to right order with the routine's parameters.
- Explicitly omitted positional argument. This is indicated by the occurrence of "(" or "," followed immediately by ")" or ",". The corresponding parameter is skipped in the association of positional arguments with parameters.
- Nonpositional argument. This is an argument (or list of arguments) immediately preceded by "parameter name :=". The parameter name must be the name of a parameter of the called routine, and this use is not affected by any other declarations. The nonpositional argument (or list of arguments) is associated with the named parameter. No other argument (positional or nonpositional) can be specified for the same parameter.

Positional arguments can be used in the same call as nonpositional ones, but any positional arguments must be listed first. For example:

```
sort( arr, out_of_order : = descending, n : = 10);
```

Note that a parameter with the LIST attribute may be associated with zero or more arguments. To specify no arguments for such a parameter, simply omit it from the argument list. Otherwise, give the list of arguments nonpositionally (that is, following "parameter name :=") or positionally at the end of an argument list. In the latter case, arguments must either be given or explicitly omitted for all the parameters preceding the LIST parameter (which is always the last parameter).

Calls to Predeclared Routines

Predeclared routines must be called exactly as shown in this manual. The calling (argument-passing) conventions used by VAXELN Pascal routines are discussed in the section "Calling Conventions," at the end of this chapter.

Parameters and Argument Passing

This section is concerned with the relationship between a parameter and the corresponding argument in a routine call. Individual subsections cover the three general types of parameters: VAR parameters, value parameters, and procedural parameters.

Additional subsections cover the rules for conformant parameters (value or VAR), OPTIONAL parameters (VAR or procedural), and LIST parameters (any type).

VAR Parameters

The argument corresponding to a VAR parameter is interpreted as a variable reference. It must be an addressable reference, and its data type must be compatible with the parameter's, as described below. Within the called routine, the VAR parameter denotes the data item established by the interpretation of the argument.

Using a VAR parameter in effect renames the argument within the called routine. Accessing the parameter's value accesses the argument's current value. Modifying the parameter modifies the argument. However, if the argument is actually a READONLY data item, modification is an error, with unpredictable effects.

Note that interpretation of the argument establishes the argument data item (but not its value) for the duration of the called routine.

For example:

```
VAR a: ARRAY[1..100] OF INTEGER;
    n: INTEGER;

PROCEDURE p(x,i: INTEGER);
  BEGIN i := 3;
        x := x + 1;
  END;

n := 2;
p(a(n),n);
```

During execution of p, the parameter x denotes the second element of the array a, while i denotes the variable n. Execution of p assigns 3 to n (via i), but this does not affect the running of the assignment statement x := x + 1, which increments the second element of a.

Type Compatibility for VAR Parameters and Arguments

The data types of a VAR parameter and its argument must be equivalent, with the following exceptions:

- If the parameter's type is ANYTYPE, the argument may be of any type.
- If the parameter's type is BYTE_DATA(*n*), the argument is implicitly typecast to BYTE_DATA(*n*) before applying the equivalence requirement. In other words, the argument is compatible if its storage size is *n* bytes.

The parameter's type is determined after the value of any conformant extents are determined. In typical usage, this makes the parameter and argument types equivalent. For example, the VAR parameter type BYTE_DATA(<*n*>) is compatible with any addressable argument.

Value Parameters

The argument corresponding to a value parameter is interpreted as an expression. The resulting value is converted (if necessary) to the data type of the parameter. The converted value becomes the parameter's initial value (non-READONLY value parameter) or value (READONLY value parameter). The data types of the parameter and argument must be compatible, as described below.

A non-READONLY value parameter is a local variable of the called routine. It is initialized to the converted value of the argument, but there is no other connection between the argument and the parameter.

For example:

```
VAR n: INTEGER;

PROCEDURE p(a: INTEGER; VAR b: INTEGER);
   BEGIN  b : = 0;
          a : = a + 1;
   END;

n : = 2;
p(n,n);
```

During execution of p, the parameter a is a local variable initialized to 2 (the value of n at the time of the call), and b is a VAR parameter denoting the variable n. The assignment b : = 0 sets n to 0, but has no effect on a. The assignment a : = a + 1 sets a to 3, but has no effect on n.

READONLY Value Parameters

A value parameter is considered READONLY in the following cases:

- The parameter is declared with the READONLY attribute.
- The parameter is declared with the LIST attribute.
- The parameter is used in an extent expression in another parameter's type or in the function result type in the same routine heading.

Also, a conformant extent is treated like a READONLY value parameter, except that its value is determined by conformance rather than an explicit argument in the argument list.

A READONLY value parameter denotes the same value throughout execution of the called routine. It is an error to modify a READONLY value parameter. The compiler issues an error message for any explicit attempt to modify a READONLY value parameter; for

example, by using it as the target in an assignment statement. Modifications not detectable by the compiler (for example, via a pointer to the parameter) result in unpredictable behavior.

Type Compatibility for Value Parameters and Arguments

In general, the data type of a value parameter's argument must be assignment compatible with the parameter's data type. (See Chapter 7, "Pascal Statements," for assignment compatibility rules.)

If the types are compatible but not equivalent, the argument value is converted to the target type according to the rules for assignment statements; for example, an INTEGER argument is converted to agree with a REAL parameter.

The parameter's type is determined after the values of any conformant extents are determined. For typical conformant array types, this makes the parameter and argument types equivalent, and therefore compatible. To provide additional flexibility in the use of strings and BYTE_DATA, the following special rules apply:

- If a value parameter's type is conformant STRING(<*n*>) or VARYING_STRING(<*n*>), the argument may be any string value, including values of type CHAR and PACKED ARRAY [1..*n*] OF CHAR. The length of the argument becomes the value of the conformant extent *n*, and there is no conversion (that is, no truncation or padding with blanks).
- If a value parameter's type is conformant BYTE_DATA(<*n*>), the value may be of any type. It is converted to BYTE_DATA as explained under the CONVERT function (see Chapter 9, "VAXELN Routines"), and *n* is the length of the

converted result. In most cases, no real conversion occurs, and *n* is simply the storage size of the argument. However, PACKED ordinals and PACKED small sets may be expanded in size, and set expressions may be truncated (see the CONVERT function).

- If a value parameter's type is nonconformant BYTE_DATA(*n*), the argument may be of any type, provided that the length after conversion to BYTE_DATA is equal to *n*. It is a range violation if this is not the case. (Again, see the CONVERT function for more information.)

Argument Copying and Use of READONLY

Passing an argument to a value parameter generally (but not always) involves moving the argument's value from one storage location to another. This discussion explains why this happens and presents some guidelines for declaring parameters so as to avoid performance degradation related to argument copying.

There are two related reasons for copying the argument value. The first applies to non-READONLY value parameters. Because such a parameter is a local variable of the called routine, the value must be moved to initialize it. The called routine cannot use the value's original storage as the local variable, because that might modify data belonging to the calling routine. This situation is illustrated by the example given earlier under "Value Parameters."

The second reason for copying an argument value is to capture it, so it will not be modified by side effects of the called routine. Copying to initialize a non-READONLY value parameter accomplishes this purpose also, but the need to capture a value may specifically cause a

copy to be performed in the case of a READONLY parameter.

For example:

```
VAR x: DOUBLE;

PROCEDURE p(a: [READONLY] DOUBLE;
   VAR b: DOUBLE);
   BEGIN  b : = 3.2;
          b : = a + b;
   END;

x : = 2;
p(x,x);
```

Here, the value parameter a gets the value 2 (from x), while the VAR parameter b denotes the variable x. The assignment b : = 3.2 must change x but not a. Hence, the call to p has to copy x's value into safe storage for use as a.

The compiler does not copy argument values in cases where it can determine that avoiding the copy will not change the routine's behavior (assuming correct use of READONLY). This does not, however, eliminate all unnecessary instances of argument copying (unnecessary given the knowledge of the programmer regarding the routine's behavior, that is). Following these guidelines will generally yield the best results:

- Unless a value parameter is truly needed as a local variable, declare it with the READONLY attribute. This is good practice in any case, because the compiler will then detect inadvertent attempts to modify it.
- Performance degradation due to argument copying is not an issue for small arguments, especially for those data types that are passed immediately in

the argument list. (See "Calling Conventions," later in this chapter, for more information.)

- Array and record input parameters should generally be declared as VAR parameters, unless use of the parameter as a local variable is necessary. You must then avoid side effects that can inadvertently change the parameter's value while the called routine still needs the original value.
- Input string parameters are generally best declared as [READONLY] STRING(<*n*>). In some cases, however, it works well to use a single type of string variable throughout a program (for example, VARYING_STRING(80)), and make all parameters VAR parameters of this type. If this is done, consider declaring string constants as READONLY variables of the same type.

Procedural Parameters

A procedural parameter is declared by the occurrence of a procedure or function heading in a routine's parameter list. The argument corresponding to such a parameter must itself be the name of a procedure or function. The argument name may be one declared by a normal procedure or function declaration, or the argument may itself be a procedural parameter (declared in another routine's parameter list). In either case, the declarations of parameter and argument must be compatible in the sense explained below.

Within the called routine, a procedural parameter denotes the same routine as the corresponding argument. If the argument is the name of a procedure or function (say F) declared within another routine, one additional data item is passed as part of the argument. That item is the stack frame pointer for the current stack frame of F's parent routine. When F is invoked

(via the parameter to which it was passed), this stack frame pointer is used to locate references within F to variables and other items in its parent routine (or any other containing non-outer-level blocks).

For example:

```
PROCEDURE p;
   VAR x: ARRAY [1..100] OF REAL;
   FUNCTION f(i: INTEGER) : REAL;
   BEGIN f : = x[i];
   END;

q(f);
```

The call q(f) passes the function f to some other routine q. Any invocation of f by q (via the parameter) will return the ith element of the array x in the stack frame of p current at the point of the call q(f).

Note that procedural parameters are used heavily in the Data Access Protocol interface (module $DAP), as illustrated in the source file DAP.PAS, supplied with your development system.

The following example of a recursive "tree walk" illustrates both the use of procedural parameters and the manipulation of dynamically sized self-defining data. Here, the dynamic data structure is a tree node that can have any number of subnodes. The procedural parameter defines the action to be taken at each subnode in a complete tree.

T

P

P

Compatibility for Procedural Parameters and Arguments

A procedural parameter and its argument must be compatible in the sense that the declaration of each must specify parameter lists with the same number and types of parameters, and with the same function result type (for functions). The straightforward way to ensure compatibility is to use a procedure or function type in declaring the parameter and any routines that may be passed to it.

If the parameter and argument are not declared using the same routine type, the compiler compares the two declarations. The exact compatibility rules it uses are complicated, but are approximated by the following.

To be compatible, two procedure or function declarations must:

- Both be procedures or both be functions.
- Have compatible parameter lists.
- If functions, have equivalent result types.

To be compatible, two parameter lists must declare the same number of parameters. In addition:

- A corresponding pair of parameters must both be VAR parameters with equivalent types, value parameters with equivalent types, or procedural parameters (which must be compatible).
- If a parameter in one parameter list has the LIST, READONLY, or REFERENCE attribute, the corresponding parameter in the other list must have it also.
- If a parameter in one parameter list is conformant, the corresponding parameter in the other list must also be conformant. They must have the same

number of extents, corresponding conformant extents must occur in exactly the same positions in the two parameter lists or function result types being compared, and the extents must be passed the same way.

Conformant Parameters

A *conformant parameter* is a parameter with one or more extents that adjust automatically to the corresponding extents of the argument. The adjustable extents are called *conformant extents*, and they are implicit READONLY value parameters of the called routine. (Refer to the section on flexible types in Chapter 3, "Data Types," for background information.)

Various examples in this manual illustrate the use of conformant parameters; this subsection discusses the rules in detail.

A conformant extent is declared by using an identifier surrounded by angle brackets in place of an extent expression within the conformant parameter's type. For example:

```
PROCEDURE p(mat: MATRIX(<n>,n));
```

Here, mat is a conformant parameter of a flexible type named MATRIX. Its first extent, represented by <n>, conforms implicitly to the same extent of the corresponding argument. Notice that n can be used without the angle brackets in other extent expressions, to represent the extent value derived from <n>. For instance, if MATRIX is declared this way:

```
TYPE MATRIX(m,n: INTEGER) = ARRAY[1..m,1..n] OF
    INTEGER;
```

then the parameter mat is a "square" matrix, since the same extent value is used for both the first and second dimensions in the parameter declaration.

When two or more parameters are specified by the same type containing a conformant extent, the first parameter is conformant. The other parameters use the value derived from the first. Similarly, a procedure can declare a parameter with a fixed (per call) but unspecified number of bytes of uninterpreted data:

```
PROCEDURE p(bytes: BYTE_DATA(<n>));
```

or a character string with fixed but unspecified length:

```
PROCEDURE p(chars: STRING(<n>));
```

or a character string with fixed but unspecified maximum length:

```
PROCEDURE p(vchars: VARYING_STRING(<n>));
```

Parametric extents can be specified explicitly rather than as conformant extents:

```
PROCEDURE p(mat: MATRIX(n,n); n: INTEGER;
   vstring: VARYING_STRING(m); m: INTEGER);
```

This feature is illustrated by the tree-walk example given earlier under "Procedural Parameters."

The rules for conformant parameters and extents are as follows:

- There can be only one defining instance of a conformant extent per parameter list using the angle-bracket notation.
- The type of a conformant extent is the same as the type of the corresponding extent parameter in the flexible type.
- Conformant extents and ordinary value parameters can be used as extents in other parameter's types or in the function result type; that is, they can be terminal operands of extent expressions within those types.

- Conformant extents and value parameters used as extents are readonly value parameters within the routine; they cannot be used as local variables (that is, cannot be assigned to) within the routine's body.
- The name of a conformant extent can occur as the name of a value parameter. In this case, the conformant extent value is placed in the VAX argument list at the position of the value parameter. (Normally, all extents defined by a conformant parameter are collected in a descriptor.) This feature is primarily for defining routines written in other languages. When this feature is used, the value parameter must be declared with a type compatible with the conformant extent's and without a default value. The value parameter cannot be explicitly referenced by the nonpositional argument notation, and it is ignored for positional matching.
- The LIST attribute, OPTIONAL attribute, and default expressions are not allowed on parameters that define conformant extents.

Conformance Rules

Given a conformant value or VAR parameter and the corresponding argument, the compiler determines the value of the parameter's conformant extents by matching them with extents in the argument's type. This is to give the conformant extents values such that the resulting parameter type will be compatible with the argument type.

The compiler does not require that the parameter and argument have exactly the same flexible type. Instead, it uses a flexible set of rules whose aim is to accommodate most cases where the two types could be

considered equivalent. For example, a conformant matrix parameter will conform to an argument explicitly declared as a two-dimensional array with indices of type CHAR.

There are special rules for conformant BYTE_DATA parameters and conformant string value parameters; these are given in the subsections on VAR and value parameters. In other cases, the original parameter type (with symbolic conformant extents) and the argument type match the conformant extents in the parameter's type with the corresponding extents in the argument's type. These cases are described below.

Case 1, Same Predeclared Flexible Type. In this case, the argument and parameter types are the same predeclared flexible type, either STRING, VARYING_STRING, or BYTE_DATA. The argument's extent is the parameter's extent.

Case 2, Same User-Defined Flexible Type. In this case, the argument and parameter types are the same user-defined flexible type. A conformant extent in the parameter type gets the value of the same extent in the argument type.

Case 3, Argument Derived from Parameter. In this case, the argument and parameter types are different flexible types, but the argument type is derived from the parameter type. Here, the argument type is expanded in terms of the parameter type, to determine which extent applies. For example:

```
TYPE matrix(m,n: INTEGER) = ARRAY[1..m, 1..n] OF
   INTEGER;

TYPE squarematrix(s: INTEGER) = matrix(s,s);

PROCEDURE p(mat: matrix(<x>,x)); ...

VAR sqr: squarematrix(5);
```

```
BEGIN ... p(sqr); ... END.
```

The type of argument sqr, squarematrix(5), expands to matrix(5,5); the first extent of sqr thus is used as the extent matching <x> in the parameter mat. Note that the two types must be equivalent after the expansion; for example, they would not be equivalent if the type of parameter mat was matrix(<x>,2).

Case 4, Parameter Derived from Argument. In this case, the types are different flexible types, but the parameter type is derived from the argument type. The compiler expands the parameter type in terms of the argument type; the extents must then match symbolically. A conformant extent in the parameter type must occur as at least one of the extents of its expanded type; it then gets the value of the corresponding extent in the argument type. For example (with the same flexible types as in case 3):

```
PROCEDURE p(sqr : squarematrix(<x>)); ...

VAR mat: matrix(5,5);

BEGIN ... p(mat); ... END.
```

Here, the value parameter sqr is expanded back to a common type, matrix(x,x); x corresponds to the extent 5 in the argument mat's type, so the extents of the parameter also are 5 and 5. Notice that the argument-passing would fail (with a range violation) if the argument was

```
VAR mat: matrix(5,10);
```

because the parameter's expanded type, matrix(5,5), is not equivalent to the argument's. In addition, the conformance would fail if type squarematrix was defined as

```
TYPE squarematrix(s: INTEGER) = matrix(2*s,2*s);
```

because the expansion, matrix(2*x,2*x), does not match the argument's type symbolically.

Case 5, All Other Situations. If cases 1–4 do not apply (for example, different flexible types are involved, not derived from each other, or the argument does not have a flexible type), both types are expanded to completely eliminate user-defined flexible types. The resulting expanded types must then be equivalent by the usual type equivalence rules.

Conformance is possible in case 5, but the expanded parameter type must be one of these forms:

- *flex*(extent), where *flex* is STRING, VARYING_STRING, or BYTE_DATA
- ARRAY[*dimension1*,...*dimensionN*] OF *flex*(extent)
- ARRAY[*dimension1*,...*dimensionN*] OF *type*, where *type* is not an array type or a flexible type

The expanded argument type must also have the same form. Each conformant extent *x* must occur either as "extent" in *flex*(extent) or in one of the dimensions in one of the following forms:

1. constant-lower-bound .. *x*
2. nonconstant-lower-bound .. *x*
3. *x* .. any-upper-bound

With form 1, *x*'s ordinal value is determined so that the parameter will have the same number of elements as the argument in this dimension. In 2 and 3, *x*'s ordinal value is determined by the corresponding bound. In all cases, it is a range violation if this ordinal value is inconsistent with the type of *x*.

For example:

```
TYPE matrix(m,n: INTEGER) = ARRAY[1..m,1..n] OF
   INTEGER;

TYPE vector(l: INTEGER) = ARRAY[1..l] OF INTEGER;

TYPE vecarray(j,k: INTEGER) = ARRAY[0..J] OF
   vector(k);

VAR vec2: vecarray(5,10);

PROCEDURE p(mat: matrix(<x>,10));

BEGIN ... p(vec2); ... END.
```

The expansion of the parameter mat's type is:

```
matrix(x,10) → ARRAY [1..x,1..10] OF INTEGER
```

The expansion of the argument vec2's type is:

```
vecarray(5,10) →
ARRAY [0..5] OF vector(10) →
ARRAY[0..5] OF ARRAY[1..10] OF INTEGER ≡
ARRAY[0..5,1..10] OF INTEGER;
```

The value 6 is used for the extent x, because there are six elements in the corresponding dimension of the expanded type of the argument vec2.

ISO Conformant Extents

The ISO Pascal standard provides a different means of defining conformant array parameters. Instead of using a bound flexible type with conformant extents as a parameter's type (MATRIX(<m>, <n>)), you write a kind of explicit array type in which the conformant extents' names occur as the low and high index values in each dimension. The extent types are specified by ordinal type names embedded in the array type, as shown in Figure 8-12.

In VAXELN Pascal, this is interpreted by considering the conformant type first, as though it introduced a uniquely named flexible type, T, with the indicated extent parameters, such as p_1, p_2, ... p_n. The parameter type is then interpreted as:

```
T(<p1>, <p2>, ... <pn>)
```

For example,

```
PROCEDURE r(
   a,b : ARRAY[m..n : INTEGER; c..d : CHAR] OF REAL
   );
```

achieves the same effect as

```
TYPE
   T(m,n: INTEGER; c,d: CHAR) =
    ARRAY[m..n, c..d] OF REAL;
PROCEDURE r(
   a,b : T(<m>,<n>,<c>,<d>)
   );
```

The following rules and notes apply to an ISO conformant type:

- A distinct name must be used for each low and high index in the conformant array. Like ordinary conformant extents, these are "true" value parameters and cannot be modified within the routine.
- The name of an ISO conformant extent must not duplicate the name of any other parameter or conformant extent.
- In most cases, the use of ISO conformant arrays involves more overhead than the use of normal VAXELN Pascal conformant parameters. We recommend that ISO conformant arrays only be used in programs designed to conform to the ISO standard.

- The VAXELN Pascal interpretation of conformant arrays is not as strict as the ISO standard. It is wise to check that the declaration and usage conform strictly to the standard.

OPTIONAL VAR and Procedural Parameters

A VAR or procedural parameter may be declared with the OPTIONAL attribute, which means that an argument need not be supplied for the parameter when the routine is called. As explained in the section "Procedure and Function Calls," earlier in this chapter, arguments can be omitted by giving a short argument list, by using the nonpositional notation, or by omitting a positional argument while including the following ",".

The OPTIONAL attribute is incompatible with the LIST attribute, and it cannot be applied to a conformant parameter.

Within the called routine, the predeclared function PRESENT can be used to determine if an OPTIONAL parameter was actually passed an argument. The function

```
PRESENT(optional_parameter)
```

returns TRUE if the argument was supplied; otherwise, it returns FALSE.

It is an unpredictable error to reference an OPTIONAL parameter whose argument was not supplied, except using the PRESENT function or in passing the parameter as an argument to an OPTIONAL parameter in another routine call. For example:

```
PROCEDURE q(VAR b: [OPTIONAL] INTEGER);
  BEGIN;
    IF PRESENT(b) THEN b : = 1;
  END;
```

```
PROCEDURE p(VAR a: [OPTIONAL] INTEGER);
  BEGIN;
  .
  .
  q(a);
  END;
```

Here, the code for procedure p does not have to explicitly test for the presence of the OPTIONAL argument, because a is simply passed along as an OPTIONAL argument to q.

The following example shows how you might organize a routine that has an OPTIONAL procedural parameter:

```
PROCEDURE option_type; PROCEDURE_TYPE;
  { These are any procedures without arguments. }

PROCEDURE graph(
  VAR y_array: arraytype;
  [OPTIONAL] PROCEDURE option OF TYPE
    option_type);

PROCEDURE default OF TYPE option_type;
  BEGIN...END; { Body of default. }
  { Invoked if no option specified. }

BEGIN { Graph code. }
  IF PRESENT(option) THEN option { Execute option
  procedure. }
  ELSE default; { Use default procedure. }
```

Here, the procedural parameter option specifies some sort of initialization action related to graph. If a call to graph omits the argument option, the code for graph invokes the internal procedure default rather than option.

LIST Parameters

A parameter may be declared with the LIST attribute, which means that it may be supplied zero or more arguments. The number of arguments supplied is limited by the maximum number of arguments (253) allowed in a complete argument list.

A LIST parameter may be a value parameter, a VAR parameter, or a procedural parameter. Only one parameter of a routine can have this attribute, and it must be the last parameter in the routine's parameter list. (This rule prevents ambiguity in interpreting the argument list.) LIST is incompatible with the OPTIONAL attribute and with the specification of a default value for a value parameter. A LIST parameter must not be conformant.

Within the called routine, the LIST parameter must only be referenced as an argument of the predeclared ARGUMENT and ARGUMENT_LIST_LENGTH functions, which reference individual arguments passed to the LIST parameter and return the total number of such arguments, respectively. For example:

```
PROGRAM c(OUTPUT);
FUNCTION comparesum(
   sum : INTEGER : = 0;
   addends: [LIST] INTEGER
   ) : BOOLEAN;
   { This function returns TRUE if the argument for
   sum equals the sum of the addends. }
   VAR tempsum, addcount: INTEGER : = 0;
   BEGIN
     FOR addcount : = 1 TO
       ARGUMENT_LIST_LENGTH(addends) DO
       tempsum : = tempsum +
       ARGUMENT(addends,addcount);
```

```
        comparesum := tempsum = sum;
    END; { comparesum }
BEGIN
    IF comparesum(5,2,3) THEN
      WRITELN('5 = 2 + 3');
    IF comparesum(addends := 2,3, sum: = 5) THEN
      WRITELN('2 + 3 = 5');
    IF comparesum THEN
      WRITELN('0 = 0');
END.
```

Here, the function comparesum can accept up to 252 integer arguments for addends, since the parameter sum also is specified. Notice that although a LIST parameter like addends must be the last *parameter*, it need not be the last *argument*. The nonpositional argument form can be used to associate particular values in the call with the LIST parameter.

Calling Conventions

This section explains the conventions used to pass arguments to VAXELN Pascal procedures and functions and to return function results. The rules for procedures without conformant parameters are given first, followed by the rules for functions and conformant parameters. This section does not apply to predeclared or in-line routines, which are expanded into in-line code and, therefore, have no argument list in the ordinary sense.

Procedures

A procedure is invoked by a VAX CALL instruction with a standard VAX argument list, as shown in Figure 8-13.

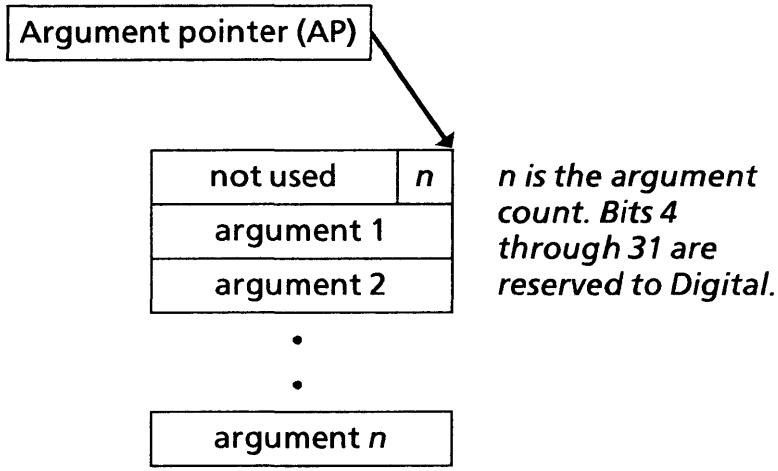


Figure 8-13. An Argument List

The VAX arguments correspond to the procedure's parameters in the order the parameters are written in the procedure's heading. The VAX arguments, if any, corresponding to a LIST parameter are at the end of the argument list. The total length of the argument list can be obtained with the ARGUMENT_LIST_LENGTH function. The total number of arguments in the list can be obtained with TOTAL_ARGUMENT_COUNT.

The contents of a particular VAX argument's longword are determined as described below.

VAR Parameter

For a VAR parameter, the VAX argument is the address of the VAXELN Pascal argument. If the parameter is optional and its VAXELN Pascal argument is omitted, the VAX argument is zero.

Procedural Parameter

The VAX argument is the address of a quadword, the first longword of which contains the address of the VAXELN Pascal argument's entry mask. (The VAXELN Pascal argument is a routine name or another procedural parameter, from which the entry mask address is obtained.)

If the VAXELN Pascal argument denotes an internal procedure or function (that is, one contained inside another routine), the second longword contains a pointer to a stack frame for its parent routine. If the procedural parameter is optional and its VAXELN Pascal argument is omitted, the VAX argument is zero.

Value Parameter

In general, if a value parameter's data type is represented in 32 or fewer bits, its argument is passed immediately. That is, the VAX argument contains the value of the VAXELN Pascal argument. The value is zero-extended to a longword (32 bits) in cases where the parameter's data type has a normal representation less than a longword (for example, type CHAR). The data types for which values are passed immediately are BOOLEAN, CHAR, INTEGER, enumerated, pointers, small sets (ORD(maxelement) < 32), REAL, and the system object types except PORT, excluding types with the BIT attribute.

If a value parameter's type is not one for which values are passed immediately, or if the value parameter would normally be passed immediately but has a REFERENCE attribute (see "The REFERENCE Attribute," later in this section), the VAX argument is the address of storage containing the VAXELN Pascal argument's value.

If a value parameter is optional and its VAXELN Pascal argument is omitted, the VAX argument is the parameter's default value (passed immediately or not, as described above).

When a value argument is passed by address, the following rules apply to the storage addressed by the VAX argument:

- If the parameter has the READONLY or LIST attribute, the argument's storage must be such that it will not be modified while the called routine is executing. For example, it must not be the storage occupied by a variable that will be modified by the called routine. (The called routine will not directly modify this storage.)
- If the parameter is not READONLY or LIST, but is a record or array whose boundary requirement is only bit alignment, the argument storage must be a dummy variable that can be freely used by the called routine.
- For all other value parameters, the only requirement is that the argument's storage contain the argument's value when the call is executed. The called routine copies the value into a local variable before executing any code that could change the contents of the argument's storage.

Function Results

Function results are returned as follows:

1. If the function's result data type is one for which value arguments are passed immediately, the function's result is returned in register R0.

2. If the function's result data type is LARGE_INTEGER or DOUBLE, the function result is returned in the register pair R0, R1.
3. Otherwise, the function result is returned in storage whose address is given by the first VAX argument. This storage is used as a local variable during the function's execution. Note that the position of each VAXELN Pascal argument in the VAX argument list is displaced by one longword in this case.

Conformant Parameters

The argument corresponding to a conformant parameter is normally passed by a descriptor. The VAX argument is the address of the descriptor. The descriptor contains the address of the data and the conformant extent values.

The first extent value is the descriptor's first longword; the second longword contains the address of the data. Subsequent longwords contain any further extent values, one per longword. The extent values are treated as 32-bit integers; that is, ORD(extent) is stored in the descriptor.

The following notes apply:

- The data item denoted by the address in the descriptor is determined in accordance with the rules for nonconformant parameters.
- If a conformant extent is also named explicitly in the routine's parameter list, it will not be in the descriptor. If all conformant extents of a parameter are so named, the argument is passed by address, not by descriptor.
- The form of descriptor used is simpler than the VAX standard descriptor. In general, passing

arguments by descriptor to an external routine (not called in VAXELN Pascal) has to be treated in VAXELN Pascal as passing a record with the appropriate contents. However, the VAXELN Pascal descriptors for STRING(<n>) and VARYING_STRING(<n>) have the same form as the standard VAX descriptors for strings. VAXELN Pascal accepts the standard form.

A non-VAXELN Pascal routine will accept the VAXELN Pascal form unless it accesses the second word of the descriptor, which, in a standard VAX descriptor, contains a code denoting the particular descriptor class.

The REFERENCE Attribute

A value parameter may be declared with the REFERENCE attribute, which means that its argument will be passed by address rather than immediately in the VAX argument list. The REFERENCE attribute can only be specified for value parameters whose data type is such that they would otherwise be passed immediately in the argument list.

Chapter 9
VAXELN Routines

Introduction

VAXELN Pascal supplies procedures and functions that perform commonly used operations. Many of the routines are predeclared; these routines must be called exactly as shown in the call format for each routine.

Most of the operations on the system data types are performed by predeclared kernel procedures, referred to as *kernel services*. These services create, delete, or otherwise affect the state of the kernel objects represented by the system data types. (See Chapter 3, "Data Types," for a description of the system data types.)

Other VAXELN Pascal routines are not predeclared. These routines are used in programs by including particular modules from the library RTLOBJECT.OLB in the compilation.

This chapter describes all VAXELN Pascal routines that do not fall under specific topics discussed in later chapters. These routines are categorized as follows:

- Arithmetic functions
- Ordinal functions
- String functions
- Type conversion routines
- Argument functions
- Storage allocation routines
- VAX functions

- Time representation routines
- Other routines

Within each category, the functions and procedures are listed in alphabetical order. A brief description of each routine is given, followed by the call format and the arguments and function result (if appropriate). Note that functions always return a value that is associated with the function identifier.

The other VAXELN routines are located as follows:

- Chapter 10, "Queues," describes the procedures relating to queues.
- Chapter 11, "Subprocesses and Synchronization," describes the kernel services relating to processes and synchronization, the authorization procedures, the Authorization Service utility, program loader utility, and exit utility procedures, and the mutex procedures.
- Chapter 12, "Interjob Communication," describes the kernel services relating to message transmission and interjob data sharing, the memory allocation procedures, and the stack utility procedures.
- Chapter 13, "Errors and Exception Handling," describes the procedures relating to exception handling.
- Chapter 14, "Device Drivers and Interrupts," describes the kernel services relating to devices, the IPL procedures, the DMA device handling procedures, the device register procedures, and the real-time device driver utility procedures.
- Chapter 15, "Input and Output," describes all Pascal I/O routines, as well as the VAXELN file utility, disk utility, and tape utility procedures.

Arithmetic Functions

The arithmetic functions described in this section perform mathematical computations. Table 9-1 summarizes these functions.

Table 9-1. Arithmetic Functions

Function	Purpose
ABS(x)	computes the absolute value of x.
ARCTAN(x)	computes the arctangent of x.
COS(x)	computes the cosine of x.
EXP(x)	computes the exponential of x.
LN(x)	computes the natural logarithm of x.
ODD	indicates whether an integer is odd or even.
SIN(x)	computes the sine of x.
SQR(x)	computes the square of x.
SQRT(x)	computes the nonnegative square root of x.
XOR	performs an exclusive OR.
ZERO	sets a variable to binary zero.

ABS

The ABS function returns the absolute value of its argument.

Call Format

```
RESULT : = ABS(expression)
```

Arguments and Result

expression. The argument supplies a value of type INTEGER, LARGE_INTEGER, REAL, or DOUBLE.

The result is the expression's absolute value; its data type is the same as the expression's.

ARCTAN

The ARCTAN function returns the arctangent of its argument.

Call Format

```
RESULT : = ARCTAN(expression)
```

Arguments and Result

expression. The argument supplies an expression of type REAL, DOUBLE, or INTEGER. An INTEGER argument is converted to REAL before the result is computed.

The result is a REAL or DOUBLE value θ such that $-\pi/2 < \theta < \pi/2$. That is, θ is the angle in radians whose tangent is expression.

COS

The COS funtion returns the cosine of its argument.

Call Format

```
RESULT : = COS(expression)
```

Arguments and Result

expression. The argument supplies an expression of type REAL, DOUBLE, or INTEGER, representing an angle in radians. An INTEGER expression is converted to REAL before the computation is performed.

The result is a REAL or DOUBLE value representing the cosine.

EXP

The EXP function raises the base of the natural logarithms (*e*) to a specified power.

Call Format

```
RESULT : = EXP(power)
```

Arguments and Result

power. The argument supplies an INTEGER, REAL, or DOUBLE exponent. An INTEGER argument is converted to REAL before the computation is performed.

The result is the REAL or DOUBLE value e^{power}.

LN

The LN function returns the natural (base *e*) logarithm of its argument.

Call Format

```
RESULT : = LN(value)
```

Arguments and Result

value. The argument supplies an INTEGER, REAL, or DOUBLE value, which must be greater than zero.

The floating-point result is the natural logarithm of value. If value is INTEGER or REAL, the result is REAL; if it is DOUBLE, the result is DOUBLE.

ODD

The ODD function determines whether an integer is odd or even.

Call Format

```
RESULT : = ODD(expression)
```

Arguments and Result

expression. The argument supplies an integer expression.

The result is the BOOLEAN value TRUE if expression is odd, otherwise FALSE. That is, the result is equivalent to:

```
(ABS(expression) mod 2 = 1)
```

SIN

The SIN function returns the sine of its argument.

Call Format

```
RESULT := SIN(expression)
```

Arguments and Result

expression. The argument supplies an INTEGER, REAL, or DOUBLE value representing an angle in radians. INTEGER is converted to REAL before the operation is performed.

The floating-point result (REAL or DOUBLE) is the sine of expression.

SQR

The SQR function returns the square of its argument.

Call Format

```
RESULT := SQR(expression)
```

Arguments and Result

expression. The argument supplies an integer or floating-point expression.

The result is the square of expression and has the same data type as expression.

SQRT

The SQRT function returns the nonnegative square root of its argument.

Call Format

```
RESULT : = SQRT(expression)
```

Arguments and Result

expression. The argument supplies an integer or floating-point expression; the value must not be negative.

The floating-point result is the square root of expression.

XOR

The XOR function performs an exclusive OR on two sets or BOOLEAN values.

Call Format

```
RESULT : = XOR(a,b)
```

Arguments and Result

a, b. The arguments supply two sets with the same base type or two BOOLEAN expressions.

For two set arguments seta and setb, the result is

```
(seta - setb) + (setb - seta)
```

where − and + are the set difference and union operators, respectively. That is, the result is a set of the same base type as the arguments and whose elements are the elements present in one argument but not both.

For example,

```
VAR
   seta, setb, xset: SET OF CHAR;
BEGIN
   seta : = ['a','b','c'];
   setb : = ['a','c','e'];
   xset : = XOR(seta,setb);
END.
```

makes xset = ['b','e'].

For two BOOLEAN values, the result is TRUE if one expression is TRUE and the other is FALSE; otherwise, the result is FALSE.

ZERO

The ZERO function is used in assignment statements and initializers to set a variable to binary zero.

Call Format

```
RESULT : = ZERO
```

Arguments and Result

There are no arguments.

The result is that the target is set to binary zero; that is, every bit of its internal representation is cleared.

For the rules governing the use of the ZERO function in initializers, see Chapter 4, "Constants." The rules for its use in assignment statements are as follows:

- It is allowed only as the entire right-hand side of the assignment statement.
- The target can be a variable or, within the body of a function, the function's name.

- The target must be aligned at least on a byte boundary (that is, it cannot be a packed field beginning at an arbitrary bit position, for example).
- The target must not be bit-sized (that is, it must be a whole number of bytes) unless it is an entire variable.
- The use of ZERO is a range violation unless the size of the target is less than or equal to 65,535 bytes.

Ordinal Functions

The ordinal functions described in this section require an actual parameter of an ordinal type and return a value of the same type. Table 9-2 summarizes these functions.

Table 9-2. Ordinal Functions

Function	Purpose
PRED(x)	returns the value that immediately precedes x in the ordered sequence of values of its type.
SUCC(x)	returns the value that immediately succeeds x in the ordered sequence of values of its type.

PRED

The PRED function returns the predecessor of its ordinal argument. For example, PRED('b') is 'a'.

Call Format

```
RESULT : = PRED(expression)
```

Arguments and Result

expression. The argument is an expression of ordinal type. The call is a range violation if the value of expression is the minimum value of its type.

The result is a value of the same data type as expression and with an ordinal value that is one less than expression's.

SUCC

The SUCC function returns the ordinal data item succeeding its argument. For example, SUCC('a') is 'b'.

Call Format

```
RESULT : = SUCC(expression)
```

Arguments and Result

expression. The argument supplies an expression of ordinal type; the call is a range violation if the expression result has the maximum ordinal number of its type.

The result has the same data type as expression and has an ordinal number one greater than expression's.

String Functions

The string functions described in this section manipulate character strings. Table 9-3 summarizes these functions.

Table 9-3. String Functions

Function	Purpose
FIND_MEMBER	finds the first character in a string that is a member of a specified set.
FIND_NONMEMBER	finds the first character in a string that is *not* a member of a specified set.
INDEX	returns the position of a substring.
LENGTH	returns the current length of a string.
SUBSTR	returns or refers to a substring of a specified string or BYTE_DATA expression or variable.
TRANSLATE_STRING	replaces occurrences of old characters with corresponding translation characters and returns the resulting string.

FIND_MEMBER

The FIND_MEMBER function finds the first character in a string that is a member of a specified set.

Call Format

```
RESULT : = FIND_MEMBER(
    string,
    charset
    )
```

Arguments and Result

string. This argument supplies a string value.

charset. This argument supplies a set of characters.

The result is an integer indicating the position in string of the first character that is a member of charset. The value 1 indicates the first character in the string. The value 0 indicates that none of the string's characters are members of the set.

For example, the call

```
position := FIND_MEMBER('One at a time.',[' ']);
```

finds the position (4) of the first space. The call

```
position := FIND_MEMBER(
'sys$_normal',['A'..'Z','a'..'z']);
```

finds the position (1) of the first alphabetic character.

The complementary operation is performed by FIND_NONMEMBER.

FIND_NONMEMBER

The FIND_NONMEMBER function finds the first character in a string that is *not* a member of a specified set.

Call Format

```
RESULT : = FIND_NONMEMBER(
   string,
   charset
   )
```

Arguments and Result

string. This supplies a string value.

charset. This argument supplies a set of characters.

The result is an integer indicating the position in string of the first character that is *not* a member of charset. The value 1 indicates the first character in the string. The value 0 indicates that all of the string's characters are members of the set.

For example, the call

```
position := FIND_NONMEMBER('sys $ normal',[' ']);
```

finds the position (1) of the first character that is not a space. The call

```
position := FIND_NONMEMBER(
'sys$_normal',['A'..'Z','a'..'z']);
```

finds the position (4) of the first nonalphabetic character.

INDEX

The INDEX function returns the position of a substring.

Call Format

```
RESULT : = INDEX(
    string,
    substring
    )
```

Arguments and Result

string. The first argument supplies a string value in which the substring may occur.

substring. The second argument supplies a substring.

The result is an integer giving the position of the leftmost occurrence of substring in string. The value 1 indicates the first character in string. The value 0 indicates that substring does not occur in string, which is also the case if either argument is a zero-length string or if substring is longer than string.

Notes

Uppercase and lowercase letters are not considered equivalent in this context. For example, the call

```
position : = INDEX('Machines were mice', 's were');
```

assigns the value 8 to position. The call

```
position : = INDEX('and men were lions','s were');
```

assigns 0 to position, because the substring does not occur in the string.

LENGTH

The LENGTH function returns the length of a string value.

Call Format

```
RESULT : = LENGTH(value)
```

Arguments and Result

value. The argument supplies a string value.

The result is an integer in the range 0–32767, giving the length in characters of the string value; for a VARYING_STRING variable, this is the current length.

SUBSTR

The SUBSTR function returns or refers to a substring of the specified string.

Call Format

```
RESULT : = SUBSTR(
   string,
   position,
   length
   )
```

Arguments and Result

string. The first argument is a string expression or variable, or a BYTE_DATA expression or variable.

position. This argument is an integer expression supplying the position at which the substring begins, where 1 indicates the first character in string.

length. This optional argument is an integer expression that supplies the length of the substring. If it is omitted, SUBSTR denotes the substring beginning at position and ending at the end of string.

If the string argument is a string variable, SUBSTR yields a variable reference of type STRING(*n*), where *n* is the length of the specified substring. (In this case, if string is a VARYING_STRING value, it must have a defined value or the result is an unpredictable error.) In addition, it can be used as the target of an assignment. For example:

```
SUBSTR(string,k,1) : = '*';
{replace character at position k with *}
```

If SUBSTR is used as a term in an expression (for example, on the right-hand side of an assignment or as an operand of another string operator), the result returned is the indicated substring.

If the first argument is a BYTE_DATA variable or expression, SUBSTR yields a BYTE_DATA result.

It is a range violation if the indicated substring is not within the string. That is, the following conditions must be satisfied:

$$1 \leq \text{position} \leq \text{LENGTH(string)}$$

and

$$\text{position} + \text{length} - 1 \leq \text{LENGTH(string)}$$

and

$$0 \leq \textit{length}$$

If the first argument is a VARYING_STRING variable reference, it must have a defined value.

TRANSLATE_STRING

The TRANSLATE_STRING function, given an original character string as an argument, replaces occurrences of old characters with corresponding translation characters and returns the resulting string.

Call Format

```
RESULT : = TRANSLATE_STRING (
    original,
    translation,
    OLDCHARS : = old
    )
```

Arguments and Result

original. The first argument supplies the string expression to be translated.

translation. If old is not present, this argument is a string expression providing a translation table used to translate the characters in original. If old is present, this argument must be a string constant defined in the current module (that is, not imported from another module); in this case, translation and old together specify the desired translation.

old. This optional argument must be a string constant defined in the current module. Together with translation, it specifies the desired translation.

The result is the translated character string.

Notes

When old is present, the compiler constructs the translation table by using old's characters as indices into translation.

For example,

```
TRANSLATE-STRING(
    TEXT-LINE, 'AEIOU',OLDCHARS: = 'aeiou')
```

translates all the lowercase vowels in TEXT-LINE to uppercase and returns the resulting string.

If old is omitted, the translation argument indexes the entire 256-character character set in ascending order.A particular character in translation thus corresponds to the character at the same position in the ordinal sequence; for example, the first character in translation is the translation of CHR(0), the second character is the translation of CHR(1), and so on. It is a range violation if translation contains no translation for some character occurring in original.

Type Conversion Routines

The conversion functions described in this section convert parameters of one type to another type, returning the converted value of the new type. The procedures (PACK, UNPACK) pack and unpack array parameters. Table 9-4 summarizes these routines.

Table 9-4. Type Conversion Routines

Routine	Purpose
BIN	converts an argument to its binary representation.
CHR	converts an integer to a character.
CONVERT	converts a value to another type.
HEX	converts an argument to its hexadecimal representation.
OCT	converts an argument to its octal representation.
ORD	returns the ordinal value of its argument.
PACK	converts elements of an unpacked array to a packed array.
ROUND	rounds a floating-point value to the nearest integer.
TRUNC	truncates a floating-point number to produce an integer.
UNPACK	converts elements of a packed array to an unpacked array.

BIN

The BIN function returns a character string representing the binary value of its argument.

Call Format

```
RESULT : = BIN(
    expression,
    length,
    digits
    )
```

Arguments and Result

expression. This argument is an expression of any type. Some types are converted implicitly to BYTE_DATA, following the rules given in under CONVERT.

length, digits. These optional arguments are integer expressions supplying the total length of the result in characters and the minimum number of significant digits, respectively. If they are omitted, the length in bits of the converted expression is the default value for *digits*; the default for *length* is one greater, causing the string to be preceded by a space character.

The result is a character string whose format is: '□bbbb...' where bbbb... are the characters 0 and 1. The binary digits are preceded by a space.

CHR

The CHR function returns the character corresponding to its integer argument.

Call Format

```
RESULT : = CHR(expression)
```

Arguments and Result

expression. The argument supplies an integer expression, which must be in the range 0–255, inclusive; otherwise, the expression is a range violation.

The result is the single character ch such that ORD(ch) = expression.

CONVERT

The CONVERT function converts a value to another type.

Call Format

```
RESULT : = CONVERT(
   type,
   value
   )
```

Arguments and Result

type. This argument supplies the type to which to convert value. It can be ordinal, a set type, BYTE_DATA, LARGE_INTEGER, REAL, DOUBLE, PACKED ARRAY OF CHAR, STRING, or VARYING_STRING.

value. This argument supplies the value to be converted, which must be of type LARGE_INTEGER, REAL, DOUBLE, a set type, an ordinal type (excluding CHAR), or string type.

The result is value converted to type.

Note that the following conversions are not done by CONVERT because they are available with other routines:

- Ordinal types to integer: Use ORD.
- Integer to CHAR: Use CHR.
- REAL or DOUBLE to integer: Use ROUND or TRUNC.
- Conversion of a value's internal representation to a string: Use BIN, OCT, or HEX.

INTEGER/LARGE_INTEGER Conversions

The only conversion for type LARGE_INTEGER is to or from INTEGER (or subranges of INTEGER). Overflow occurs if a LARGE_INTEGER source is too large for the target type; otherwise, the conversion is exact. INTEGER to LARGE_INTEGER is exact.

Conversions with Ordinal Types

If an integer is converted to BOOLEAN or an enumerated type, or if the target type is one of these types and the source value is the same type, the result is a value *v* of the target type such that ORD(*v*) equals the source value. For example, the result of CONVERT(BOOLEAN,0) is FALSE, because ORD(FALSE) is 0. It is a range violation if *v* does not exist in the target type.

Only strings of length 1 can be converted to CHAR.

Conversions to or from REAL and DOUBLE

If the source value is REAL or DOUBLE, the target type must be REAL or DOUBLE. The conversion of integers to REAL or DOUBLE is exact, as is REAL to DOUBLE and REAL to REAL. The conversion of a DOUBLE value or integer to REAL is by rounding. If the source value is a floating-point constant, its decimal representation is rounded to convert it to REAL or DOUBLE.

Conversions of Strings

The conversion of a string to an ordinal (except CHAR), REAL, or DOUBLE value uses a field of the string, beginning at the first character that is not a space or tab and ending at the end of the string or an invalid character; in other words, the conversion is comparable to that performed by the READ procedure with a textfile.

Conversions to String Types

If the type argument is a string type, the length of the result is determined by the source value represented as a string. The actual length and format of the converted result are the same as for the WRITE procedure used with a textfile, except that leading spaces are not used. If the extent of the string type is less than the width of the result, the effect is the same as when a short field width is specified in a WRITE call.

The following example converts the value TRUE or FALSE to the string 'TR' or 'FA':

```
CONVERT(STRING(2),boolean_value)
```

In conversions of strings to strings, CONVERT uses the rules for padding, truncation, and length matching that apply in string assignments. For example, if the target

type is a packed array of CHAR, it is a range violation unless the source value has the same length.

Conversions with Set Types

If the type argument is a set type, the source value must be a set with the same ordinal base type. The representation rules for sets determine an integer *n*, such that the target type is represented by *n* longwords. If the source set has a larger *n*, it is truncated to the number for the target type (in other words, any elements in the source set are discarded if ORD(element) $\geq 32 \times n$). The result value is a set of the target type containing only those elements in the (possibly truncated) source value. It is a range violation if the truncated value has any elements that are not in the range of the target type.

Sets can also be converted to BYTE_DATA, as described below.

Conversions to BYTE_DATA

Any of the valid source types can be converted to BYTE_DATA. If the BYTE_DATA size is not specified, it is determined implicitly from the size in bytes of the source.

The source is always treated as a value for this conversion; for instance, if it is a function, the function is called and its result is used as the source value.

If the source value is a packed field, it is unpacked to its normal representation. If it is a bit-aligned array or record, it is copied to a byte-aligned temporary, if necessary.

If the source is a set expression more general than a variable, it is truncated, if necessary, so that its size is 32 bytes.

If an explicit size is given (for example, BYTE-DATA(8)), the source value as modified by the above rules must have the same size, or else the conversion is a range violation.

HEX

The HEX function returns a character string representing the hexadecimal value of its argument.

Call Format

```
RESULT : = HEX(
    expression,
    length,
    digits
    )
```

Arguments and Result

expression. This argument is an expression of any type. Some types are converted implictly to BYTE_DATA, following the rules given under CONVERT for that type.

length, digits. These optional arguments are integer expressions supplying the total length of the result and the minimum number of significant digits, respectively. If they are omitted, the length in hexadecimal digits of the converted expression is the default number of digits; the default for *length* is then one greater, causing the string to be preceded by a space character.

The result is a character string whose format is: '□dddd...' where dddd... are from the sets of characters 0–9 and A–F. The hexadecimal digits are preceded by a space.

OCT

The OCT function returns a character string representing the octal value of its argument.

Call Format

```
RESULT : = OCT(
    expression,
    length,
    digits
    )
```

Arguments and Result

expression. The expression can be of any type. Some types are converted implicitly to BYTE_DATA, following the rules given in CONVERT.

length, digits. These optional arguments are integer expressions supplying the total length of the result and the minimum number of significant digits, respectively. If they are omitted, the length in octal digits of the converted expression is the default value for the *digits* argument; the default for *length* is one greater, causing the string to be preceded by a space character.

The result is a character string whose format is: '□dddd...' where dddd... are the characters 0–7. The octal digits are preceded by a space.

ORD

The ORD function returns the ordinal number of its argument.

Call Format

```
RESULT : = ORD(expression)
```

Arguments and Result

expression. The argument supplies an expression of ordinal type or a string expression with one character (which is treated as a value of type CHAR). It is a range violation if the length of a string argument does not equal 1.

The result is an integer giving the ordinal number of expression.

PACK

The PACK procedure transfers elements of an unpacked array to a packed array.

Call Format

```
PACK(
   unpacked,
   first,
   packed
   )
```

Arguments

unpacked. The first argument is an unpacked array variable. If it is multidimensional, it is treated as a one-dimensional array whose elements are arrays.

first. This argument supplies an ordinal value that is the index of an element in unpacked. This element will be the first element transferred to packed.

packed. This argument is a packed array variable that receives elements from unpacked. If it is multidimensional, it is treated as a one-dimensional array whose elements are arrays. The types of its elements and those of unpacked must be equivalent. The first element of packed receives unpacked[first], and so on until packed is filled.

The range requirements are as follows:

- The lower bound of unpacked must not exceed first.
- The sum of first and the number of elements in packed must not exceed 1 plus the upper bound of unpacked:

  (first) + (no.-elements(packed))
  ≤ up-bound(unpacked) + 1

  That is, the procedure must not demand the transfer of more elements from unpacked than exist between unpacked[first] and the end of unpacked.

Algorithm

Given:

UA is a variable of type ARRAY[ITYPE1] OF T,
PA is a variable of type PACKED ARRAY[ITYPE2] OF T,
LOW and HIGH are the minimum and maximum values of ITYPE2 (the minimum and maximum indices of PA), and
FIRST is a value that is assignment compatible with ITYPE1 (that is, a valid index of array UA).

Then PACK(UA,FIRST,PA) is equivalent to

```
begin
k : = FIRST; { Index first element. }
for j : = LOW to HIGH do { For all elements of PA. }
   begin
     PA[j] : = UA[k]; { Transfer element. }
     if j <> HIGH then k : = succ(k)
       { Index next element. }
     end
end
```

where k is an auxiliary variable of type ITYPE1 and j is an auxiliary variable of type ITYPE2.

ROUND

The ROUND function rounds a floating-point expression to the nearest INTEGER value.

Call Format

```
RESULT : = ROUND(expression)
```

Arguments and Result

expression. The argument supplies a floating-point expression (type REAL or DOUBLE).

The result is the integer formed by adding or subtracting 0.5, depending on the sign of expression, and truncating the result.

For example, ROUND(3.5) is 4, ROUND(– 3.5) is – 4, and ROUND(– 3.3) is – 3.

Overflow occurs if the rounded result is not in the range of type INTEGER.

TRUNC

The TRUNC function truncates a floating-point number to produce an integer and returns the integer.

Call Format

```
RESULT : = TRUNC(expression)
```

Arguments and Result

expression. The argument supplies a floating-point expression (type REAL or DOUBLE).

The result is an integer formed by truncating the fractional part of the floating-point expression. The result has the same sign as expression. For example, TRUNC(3.5) is 3; TRUNC(– 3.5) is – 3.

Integer overflow occurs if the truncated result is outside the range of type INTEGER.

UNPACK

The UNPACK procedure transfers all the elements of a packed array to an unpacked array.

Call Format

```
UNPACK(
   packed,
   unpacked,
   first
   )
```

Arguments

packed. This argument is a packed array. If it is multidimensional, it is treated as a one-dimensional array whose elements are arrays.

unpacked. This argument is an unpacked array. If it is multidimensional, it is treated as a one-dimensional array whose elements are arrays. The types of its elements and those of packed must be equivalent. The element unpacked[first] receives the first element of packed, and so on until all the packed elements have been transferred.

first. This argument is an ordinal value that is the index of an element of unpacked. It indexes the element of unpacked that receives the first element of packed.

The range requirements are as follows:

- The lower bound of unpacked must not exceed first.
- The difference between the upper bound of unpacked and first, plus 1, must be equal to or greater than the number of elements in packed:

  (up-bound(unpacked) – first) + 1 ≥
  no.-elements(packed)

  That is, enough elements must be available in the indexed portion of unpacked to hold all the values in packed.

Algorithm

Given:

UA is a variable of type ARRAY[ITYPE1] OF T,
PA is a variable of type PACKED ARRAY[ITYPE2] OF T,
LOW and HIGH are the minimum and maximum values of ITYPE2 (the minimum and maximum indices of PA), and

FIRST is a value that is assignment compatible with ITYPE1 (that is, a valid index of array UA).

Then UNPACK(PA,UA,FIRST) is equivalent to

```
begin
k : = FIRST; { Index first element. }
for j : = LOW to HIGH do
   { For all elements of PA. }
   begin
   UA[k] : = PA[j]; { Transfer element. }
   if j <> HIGH then k : = succ(k)
   { Index next element. }
   end
end
```

where k is an auxiliary variable of type ITYPE1 and j is an auxiliary variable of type ITYPE2.

Argument Functions

The functions described in this section obtain arguments, argument list lengths, and argument counts. Table 9-5 summarizes these functions.

Table 9-5. Argument Functions

Function	Purpose
ARGUMENT	references an argument that corresponds to a LIST parameter.
ARGUMENT_LIST_LENGTH	returns the number of arguments passed to a LIST parameter.
PRESENT	indicates whether the argument list of the routine from which it is called contains an argument corresponding to the specified optional parameter.
PROGRAM_ARGUMENT	returns the character string passed as a program argument.

Table 9-4. Continued

Function	Purpose
PROGRAM_ARGUMENT_COUNT	returns the number of arguments passed to the program.
TOTAL_ARGUMENT_COUNT	returns the number of arguments passed to the current routine.

ARGUMENT

The ARGUMENT function denotes a particular argument that corresponds to a function or procedure parameter with the LIST attribute. It must be called only within a procedure or function declared with such a parameter or within a subordinate routine.

ARGUMENT is the only valid means of reference to an argument corresponding to a LIST parameter.

Call Format

```
RESULT : = ARGUMENT(
   parameter_name,
   argument_number
   )
```

Arguments and Result

parameter_name. This argument supplies the name of a parameter declared with the LIST attribute.

argument_number. This argument supplies a positive INTEGER value identifying the argument. The first argument in a particular list is denoted by 1. The total length of the argument list can be obtained with ARGUMENT_LIST_LENGTH.

It is a range violation if the value supplied for argument_number is less than 1 or exceeds

```
ARGUMENT_LIST_LENGTH(parameter_name)
```

Within the called routine, ARGUMENT denotes the corresponding argument exactly as if the parameter was not a LIST parameter and the call to ARGUMENT was, instead, the parameter name.

That is:

- If the LIST parameter is a value parameter, ARGUMENT denotes the corresponding value in the argument list.
- If the LIST parameter is a VAR parameter, ARGUMENT is a reference to the corresponding variable in the argument list.
- If the LIST parameter is a procedural parameter, ARGUMENT denotes the function or procedure passed to that parameter.

For example:

```
PROCEDURE f(
    [LIST] PROCEDURE propar(x,y: INTEGER)
    );
      BEGIN
      .
      .
      ARGUMENT(propar,3)(1,2);
      { Call third argument for propar with the
      arguments 1 and 2. }
```

Here, procedure f has a procedural LIST parameter; that is, f's arguments are zero to 253 procedures with two integer arguments each. In the body of f, the third of these arguments is called with the arguments 1 and 2.

Note that the call to the procedure itself is formed by ARGUMENT(propar ,3), followed by the usual parenthesized argument list.

ARGUMENT_LIST_LENGTH

The ARGUMENT_LIST_LENGTH function returns the number of arguments corresponding to a LIST parameter.

Call Format

```
RESULT : = ARGUMENT_LIST_LENGTH(
   parameter_name
   )
```

Arguments and Result

parameter_name. The argument supplies the name of a LIST parameter.

The result is an INTEGER value denoting the number of arguments (zero or more) corresponding to the LIST parameter.

PRESENT

The PRESENT function indicates whether the argument list of the routine from which it is called contains an argument corresponding to the specified optional parameter. It usually is used to supply a default value (or take a default action) when the argument for a VAR or procedural parameter is omitted.

Call Format

```
RESULT : = PRESENT(parameter_name)
```

Arguments and Result

parameter_name. The argument is the name of a VAR or procedural parameter with the OPTIONAL attribute. The name must be the name of a formal parameter of the function or procedure within which PRESENT is called, or else PRESENT must be called from a subroutine of that function or procedure.

The result is the BOOLEAN value TRUE or FALSE. TRUE indicates that the argument list of the containing routine contains an argument corresponding to the optional parameter. FALSE indicates that the argument was omitted.

Notes

PRESENT is not used with value parameters, because a default value must be specified in the parameter declaration if a value parameter is optional.

PRESENT can also be used in a subroutine of the routine having the optional parameter.

PROGRAM_ARGUMENT

The PROGRAM_ARGUMENT function returns the character string passed as a program argument to the current job.

Call Format

```
RESULT : = PROGRAM_ARGUMENT(position)
```

Arguments and Result

position. The argument is an integer expression that gives the position in the argument list (in CREATE_JOB or the System Builder's program description). The first position is 1.

The result is the character string passed as the argument in, for example, a CREATE_JOB call.

The result is the null string if there is no argument or if position exceeds the number of program arguments.

PROGRAM_ARGUMENT_COUNT

The PROGRAM_ARGUMENT_COUNT function returns an integer indicating the number of arguments passed to the program.

Call Format

```
RESULT : = PROGRAM_ARGUMENT_COUNT
```

Arguments and Result

There are no arguments.

The result is an INTEGER value giving the number of arguments passed.

TOTAL_ARGUMENT_COUNT

The TOTAL_ARGUMENT_COUNT function returns the number of arguments in the VAX argument list passed to the current routine. This function is intended for use within a routine that may be invoked with an argument list whose length is inconsistent with the routine's declaration. It may be used to detect the necessity for an error action, or to take default actions for missing arguments.

Call Format

```
RESULT : = TOTAL_ARGUMENT_COUNT
```

Arguments and Result

There are no arguments.

The result is an INTEGER value giving the number of arguments passed.

Notes

This function must not be used within a routine with the INLINE attribute.

To use this function successfully, the procedure calling conventions used by the VAXELN Pascal compiler must be thoroughly understood. In particular, note that if a missing argument is used in a routine's prologue code, an unpredictable error will occur before any code in the body of the routine is executed.

Storage Allocation and Address Routines

The routines described in this section allocate and release storage for variables (NEW, DISPOSE), return the address of a specified variable (ADDRESS), and return the size in bytes of a variable or type (SIZE). Table 9-6 summarizes these routines.

Table 9-6. Storage Allocation and Address Routines

Routine	Purpose
ADDRESS	returns the address of the variable reference supplied as its argument.
DISPOSE	releases storage previously allocated by NEW.
NEW	allocates storage for new variables.
SIZE	returns the size of its argument's storage.

ADDRESS

The ADDRESS function returns the address of the variable reference or routine name supplied as its argument.

Call Format

```
RESULT : = ADDRESS(variable)
```

Arguments and Result

variable. The argument supplies a variable of any type. The variable must be addressable. The argument can also be the name of a routine.

If variable is of type *t*, then the result is of type ↑*t*. The validity of the pointer returned is subject to these rules:

- If variable denotes all or part of the storage of a local variable or value parameter, the pointer is valid only in the current process and becomes invalid when the block activation to which it belongs terminates.
- If variable is a reference to all or part of a VAR parameter, the result pointer must not be used in any other process, and it must not be used after return from the routine of which the variable is a parameter.

Violation of these rules results in unpredictable errors that are difficult to diagnose.

If, instead of a variable, the argument for ADDRESS is the name of a routine, the result is of type ↑ ANYTYPE and is the address of the entry mask. This feature can be used to pass the address of a routine to an external procedure written in another language or as an argument of the INVOKE procedure.

It is an error to take the address of a predeclared routine or a routine with the INLINE attribute. A warning message is issued if you take the address of a routine that is not declared at the outer level or if you take the address of a program, process block, or interrupt service routine.

DISPOSE

The DISPOSE procedure releases the storage previously allocated for a data item by NEW. The storage is identified by a pointer previously returned by NEW; after the DISPOSE operation, the pointer value is invalid.

Call Formats

There are two call formats; Format 2 is used only with variant records allocated by a similar NEW call and is included only to support standard Pascal.

Format 1 DISPOSE(pointer)

Format 2
```
DISPOSE(
    pointer,
    tag-list
    )
```

Arguments

pointer. This argument supplies a pointer value, previously returned by the NEW procedure, to the variable whose storage is to be released. The pointer must not be NIL or invalid. The pointer can be of any type, including ↑ANYTYPE. The pointer is invalid following the DISPOSE operation.

tag-list. If the first argument denotes a variant record, this argument may supply one or more constants to

select variants, as in the NEW procedure. The compiler checks that the tags are valid in this usage, but they have no effect on the operation of DISPOSE.

Notes

The DISPOSE procedure returns the released storage to a list of free storage available for future calls to NEW in the current job. Small blocks of free storage are collected together in the list if they are contiguous; this reduces fragmentation of the job's virtual address space.

However, the DISPOSE procedure does not return released storage to the kernel for use by other jobs in the system. If this is required, ALLOCATE_MEMORY and FREE_MEMORY should be used explicitly.

NEW

The NEW procedure allocates heap storage for a data item. The new item's address is assigned to a pointer variable, and the variable's exact pointer type determines the size of the allocated item. The pointer is valid in the entire job. It becomes invalid when passed to the DISPOSE procedure.

Call Formats

There are two call formats; Format 2 is used only with variant records.

Format 1 NEW(pointer)

Format 2
```
NEW(
    pointer,
    tag-list
    )
```

Arguments

pointer. This argument is a variable reference whose type is a pointer type, excluding ↑ANYTYPE. The associated type of the pointer determines the size of the allocated item, and upon return, the address of the allocated item is assigned to the pointer variable.

tag-list. This is a list of tags (case constants) for a dynamically allocated variant record; the tags are separated by commas. The first tag must select a variant in the record type's variant part. Successive tags, if any, must select a variant in the variant part of the variant selected by preceding tags.

Notes

The values of all items allocated with NEW are initially undefined.

In Format 2, the tags cause NEW to allocate only the amount of storage required to hold the specified variant; they do not assign the listed values to the tag fields or in any other way select the variant. (See Chapter 3, "Data Types," for more information on records with variants.)

NEW allocates storage from a free space list maintained within the job's P0 memory. (Note that because P0 is shared by a job's processes, variables allocated this way are potentially shareable.) If this space list is empty, NEW uses ALLOCATE_MEMORY to allocate new memory to the job. Space can be returned to the free list by using DISPOSE.

NEW always allocates memory in 16-byte quantities aligned on quadword boundaries. A variable allocated by NEW also has an 8-byte internal header that is used by NEW and DISPOSE to save both the size of the

variable and a pointer to the next block of free memory. Therefore, if NEW is called to allocate an INTEGER variable, the new variable will occupy 16 bytes: 4 bytes for the integer, 8 bytes for the header, and 4 bytes to round up to a 16-byte multiple.

SIZE

The SIZE function returns the size in bytes of its argument's storage. This function is allowed in extent expressions, with the restrictions noted below.

Call Formats

There are two call formats.

Format 1
```
RESULT : = SIZE(
    variable_reference,
    tag-list
    )
```

Format 2
```
RESULT : = SIZE(
    named_type,
    tag-list
    )
```

Arguments and Result

variable_reference. This argument supplies a reference to an addressable variable.

named_type. This argument supplies a data type: either a declared type name or a bound flexible type. That is:

```
type-name
flexible-type-name(extent-list)
```

tag-list. If the first argument is a variant record or variant record type, this argument may supply one or more constants to select variants, as in the NEW

procedure. The constants cause the SIZE function to compute a size based only on the selected variants and not the entire record.

The result is the size of the identified item in bytes. If the item is a variant record, and no tag list is supplied, the result is the size of the largest possible variant. If the size of the data item is not an even number of bytes, it is rounded up to the next byte.

Notes

If used in an extent expression, the first argument must be an identifier denoting a type, variable, or parameter with a constant-size data type. Additional arguments, if any, must be extent expressions that select variants.

VAX Functions

The functions described in this section use VAX instructions to perform VAXELN Pascal operations. (See the *VAX Architecture Handbook* for more information on the VAX instructions themselves.) Table 9-7 summarizes these funtions.

Table 9-7. VAX Functions

Routine	Purpose
MOVE_PSL	returns the current contents of a VAX processor status longword.
PROBE_READ	indicates whether the first and last bytes of the specified variable are accessible for reading in the current processor mode.
PROBE_WRITE	indicates whether the first and last bytes of the specified variable are accessible for writing in the current processor mode.

MOVE_PSL

The MOVE_PSL function returns the current contents of a VAX processor status longword treated as a 32-bit integer. This operation is performed by the VAX MOVPSL instruction and is not affected by any compiler optimizations.

Call Format

```
RESULT : = MOVE_PSL
```

Arguments and Result

There are no arguments.

The result is an INTEGER value representing the current processor status longword contents.

PROBE_READ

The PROBE_READ function indicates whether the first and last bytes of the variable supplied as its argument are accessible for reading in the current processor mode. The probe operation is performed by the VAX PROBER instruction and is not affected by any compiler optimizations.

Call Format

```
RESULT : = PROBE_READ(variable)
```

Arguments and Result

variable. The argument must be an addressable variable reference whose size is >0 and <65K bytes.

The result is the BOOLEAN value TRUE if both bytes are accessible for reading; otherwise FALSE.

PROBE_WRITE

The PROBE_WRITE function indicates whether the first and last bytes of the variable supplied as its argument are accessible for writing in the current processor mode. The probe operation is performed by the VAX PROBEW instruction and is not affected by any compiler optimizations.

Call Format

```
RESULT : = PROBE_WRITE(variable)
```

Arguments and Result

variable. The argument must be an addressable variable reference whose size is >0 and <65K bytes.

The result is the BOOLEAN value TRUE if both bytes are accessible for writing; otherwise FALSE.

Time Representation Routines

The routines described in this section are provided for converting between time values represented by the 64-bit integer type LARGE_INTEGER and representative character strings. Table 9-8 summarizes these routines.

Table 9-8. Time Representation Routines

Routine	Purpose
GET_TIME	returns the current absolute system time.
SET_TIME	sets a new absolute system time.
TIME_FIELDS	returns a special record type, TIME_RECORD, whose fields represent the integer components of a complete 64-bit time value.
TIME_STRING	returns a character string representing its LARGE_INTEGER argument.
TIME_VALUE	returns the LARGE_INTEGER value corresponding to a character string.

Note: Negative time values represent time intervals by convention; nonnegative values are absolute times. Therefore, when you want to *add* a time interval to an absolute time, you must *subtract* the LARGE_INTEGER value representing the interval from the one representing the absolute time.

GET_TIME

The GET_TIME kernel procedure returns the current system time.

Call Format

```
GET_TIME(
   time,
   STATUS : = stat
   )
```

Arguments

time. This argument is a LARGE_INTEGER variable that receives a value representing the time of day. (The value can be converted to a character string with the TIME_STRING function.)

stat. This optional argument is an INTEGER variable that receives the completion status of GET_TIME.

Status Values

KER$_SUCCESS. The procedure completed successfully.

KER$_TIME_NOT_SET. The time of day has not been set. This is an alternate success status.

KER$_BAD_COUNT. The procedure call specified an incorrect number of arguments.

KER$_NO_ACCESS. An argument specified is not accessible to the calling program.

SET_TIME

The SET_TIME kernel procedure sets a new system time.

Call Format

```
SET_TIME(
   time,
   STATUS : = stat
   )
```

Arguments

time. This argument is a nonnegative LARGE_INTEGER, specifying an absolute system time. The integer is created with the TIME_VALUE function and an argument in absolute time format, as in:

```
SET_TIME(TIME_VALUE('01-JAN-1985 00:00:00.00'));
```

stat. This optional argument is an INTEGER variable that receives the completion status of SET_TIME.

Status Values

KER$_SUCCESS. The procedure completed successfully.

KER$_BAD_VALUE. The time value is invalid.

KER$_NO_ACCESS. An argument specified is not accessible to the calling program.

TIME_FIELDS

The TIME_FIELDS function returns a record whose fields are the integer-valued components of a 64-bit time value.

Call Format

```
RESULT : = TIME_FIELDS(tvalue)
```

Arguments and Result

tvalue. The argument is a LARGE_INTEGER value, such as the value returned by GET_TIME and TIME_VALUE.

The result is a record of the type TIME_RECORD, which is declared for you as follows:

```
TYPE
TIME_RECORD = PACKED RECORD
   year  : 0..65535;
   month : 0..65535;
   day: 0..65535;
   hour : 0..65535;
   minute: 0..65535;
   second : 0..65535;
   hundredth:  0..65535;
   END;
```

If tvalue is a time interval (negative value), the returned year and month fields are both 0, and the day field is the number of days specified by the interval (less than 10,000).

The following example writes out the individual components of a time value:

```
PROGRAM time(OUTPUT);
  VAR
    L : LARGE_INTEGER;
    T : TIME_RECORD;
  BEGIN
    GET_TIME(L);
    T := TIME_FIELDS(L);
    WITH T DO
      WRITELN(
        year,
        month,
        day,
        hour,
        minute,
        second,
        hundredth,
        );
  END.
```

Another example is a function that computes the number of seconds between two time values:

```
  [INLINE] FUNCTION seconds(
    start, end : LARGE_INTEGER) : INTEGER;
    VAR
      T : TIME_RECORD;
    BEGIN
      T := TIME_FIELDS(start-end);
      WITH T DO
        seconds := (24*60*60*day) +
          (60*60*hour) +
          (60*minute) +
          second;
    END;
```

TIME_STRING

The TIME_STRING function converts a LARGE_INTEGER value to a character string representing an absolute time or time interval.

Call Format

```
RESULT : = TIME_STRING(tvalue)
```

Arguments and Result

tvalue. The argument is a LARGE_INTEGER value, such as the value returned by GET_TIME and TIME_VALUE.

The result is a character string representing either an absolute time or a time interval:

- If tvalue is nonnegative, it represents an absolute time, and the result format is:

  'dd-mmm-yyyy□hh:mm:ss.cc'

  dd is 1–31 (day of month); mmm is JAN–DEC; yyyy is 1858–9999; □ is a space; hh is 0–23 hours; mm is 0–59 minutes; ss is 0–59 seconds; and cc is 0–99 hundredths of a second.

- If tvalue is negative, it represents a time interval, and the result format is:

  'dddd□hh:mm:ss.cc'

  dddd is a number of days from 0 to 9999; □ is a space; hh is 0–23 hours; mm is 0–59 minutes; ss is 0–59 seconds; and cc is 0–99 hundredths of a second.

TIME_VALUE

The TIME_VALUE function converts a character string to a LARGE_INTEGER representing either absolute time or a time interval, depending on the format of the string.

Call Format

```
RESULT : = TIME_VALUE(tstring)
```

Arguments and Result

tstring. The argument is a character string with one of two formats:

- *Absolute format*, written as:

  'dd-mmm-yyyy□hh:mm:ss.cc'

  dd is 1–31 (day of month); mmm is JAN–DEC; yyyy is 1858–9999; □ is a space; hh is 0–23 hours; mm is 0–59 minutes; ss is 0–59 seconds; and cc is 0–99 hundredths of a second.

- *Interval format*, written as:

  'dddd□hh:mm:ss.cc'

  dddd is a number of days from 0 to 9999; □ is a space; hh is 0–23 hours; mm is 0–59 minutes; ss is 0–59 seconds; and cc is 0–99 hundredths of a second.

If tstring is in interval format, the result is a negative LARGE_INTEGER representing a time interval. If it is in absolute format, the result is a nonnegative value representing an absolute time.

String Syntax for TIME_VALUE Arguments

You can omit parts of strings in either interval or absolute format, subject to the following rules and defaults:

- For absolute format, the function supplies the current system time fields for any you omit. For instance, if you omit the "days" (dd) part, the current day of the month is used.
- For interval format, unspecified fields default to zero. However, if a time interval is to specify 0 days, you must enter a 0 explicitly.
- In either format, you can simply omit trailing fields, with the defaults described above. However, if you omit leading fields, you must specify the punctuation marks (including at least one space where shown above).
- In either format, the string can be preceded by any number of (leading) spaces, and there can be any number of spaces where one is shown above. However, there can be no spaces inside the date or time parts of the string.

For example, the interval string '0 00:00:1' represents a time interval of one second. So does '0 : : 1', since the omitted fields default to 0.

The absolute string '15-JUL-1985' represents the current time of day on July 15th. The absolute string '15--1985 00:00:00' represents midnight on the 15th day of the current month, and so on. (Notice that when the leading "month" field is omitted, the punctuating hyphens must be supplied.)

Other Routines

The routines described in this section do not fall into any of the previous categories. Table 9-9 summarizes these routines.

Table 9-9. Other Routines

Routine	Purpose
ADD_INTERLOCKED	adds an integer value to a WORD variable.
ENTER_KERNEL_CONTEXT	executes the specified routine in the kernel processor mode.
FIND_FIRST_BIT_CLEAR	finds the first bit whose value is 0.
FIND_FIRST_BIT_SET	finds the first bit whose value is 1.
INVOKE	calls a routine identified by an ANYTYPE pointer.

Note: To use ENTER_KERNEL_CONTEXT, include the module $KERNEL from the RTLOBJECT library in the compilation of your program.

ADD_INTERLOCKED

The ADD_INTERLOCKED function adds an integer value to a WORD (16-bit) variable using an interlocked machine instruction. The operation is safe on shared data.

Call Format

```
RESULT : = ADD_INTERLOCKED(
    delta,
    word_argument
    )
```

Arguments and Result

delta. This argument supplies the value to be added to the word_argument. It must be in the range −32768 to 32767; if not, the call is a range violation.

word_argument. This argument is a variable declared with the WORD attribute. (The WORD attribute can be applied to a packed record or a field in such a record.) Whatever its data type, the function treats it as a word-length integer variable. It supplies one of the addends and receives the sum. The value of the variable after the addition determines the function result. The variable must be aligned on a word boundary; to ensure that it is, use the ALIGNED(1) attribute as well as WORD within a record. It is a run-time error if the argument is not properly aligned.

The result is 1 if the sum is positive, 0 if the sum is zero, and −1 if the sum is negative.

ENTER_KERNEL_CONTEXT

The ENTER_KERNEL_CONTEXT procedure executes the specified routine in the kernel processor mode. The specified routine is called with the specified argument list, and its completion status is returned as the completion status of this procedure. (Note that the completion status is only available if the specified routine is a function returning an INTEGER status.)

This procedure can be used to execute a particular routine in kernel mode when, for some reason, it is not desirable to execute the entire program in kernel mode. To use the procedure, you must include the module $KERNEL in the compilation.

Call Format

```
KER$ENTER_KERNEL_CONTEXT(
   status,
   routine,
   argument_block
   )
```

Arguments

status. This optional argument is an INTEGER variable that receives the completion status of routine. (This assumes that routine is a function returning an INTEGER status.)

routine. This argument is the address of the routine to be called in kernel mode; its type is ↑ANYTYPE. A suitable value can be obtained by supplying the ADDRESS function with the routine's name as the argument.

argument_block. This argument is the address of the VAX argument list to be passed to the called routine;

its type is ↑ ANYTYPE. The argument list is a block of longwords in the standard VAX format: the first byte of the first longword contains the number of arguments; the block contains an additional longword for each of the actual arguments.

FIND_FIRST_BIT_CLEAR

The FIND_FIRST_BIT_CLEAR function finds the first bit in a PACKED ARRAY OF BOOLEAN whose value is 0.

Call Format

```
RESULT : = FIND_FIRST_BIT_CLEAR(
    vector,
    start index
    )
```

Arguments and Result

vector. This argument is a variable of type PACKED ARRAY OF BOOLEAN, with an INTEGER index type. The array must be one-dimensional.

start index. This optional argument is an integer expression indexing the element at which the search starts. It must be greater than or equal to vector's lower bound and less than or equal to 1 plus vector's upper bound; otherwise, a range violation occurs. If it is omitted, the search starts at vector's first element.

The result is an integer indexing the first element containing the value 0. If no bit is 0, the result is 1 plus the upper bound of vector. If vector or the indexed part of vector has a size of 0, the result is *start index*.

FIND_FIRST_BIT_SET

The FIND_FIRST_BIT_SET function finds the first bit in a PACKED ARRAY OF BOOLEAN whose value is 1.

Call Format

```
RESULT : = FIND_FIRST_BIT_SET(
    vector,
    start index
    )
```

Arguments and Result

vector. This argument is a variable of type PACKED ARRAY OF BOOLEAN, with an INTEGER index type. The array must be one-dimensional.

start index. This optional argument is an integer expression indexing the element at which the search starts. It must be greater than or equal to vector's lower bound and less than or equal to 1 plus vector's upper bound; otherwise, a range violation occurs. If it is omitted, the search starts at vector's first element.

The result is an integer indexing the first element containing the value 1. If no bit is 1, the result is 1 plus the upper bound of vector. If vector or the indexed part of vector has a size of 0, the result is *start index*.

INVOKE

The INVOKE procedure calls a procedure or function identified by an ANYTYPE pointer.

Call Format

```
INVOKE(
    pointer,
    routine_type,
    argument-list
    )
```

Arguments

pointer. This argument is a pointer to the routine's entry mask. It's type must be ↑ANYTYPE. Such a pointer value can be obtained from the ADDRESS function or from routines written in other programming languages. The identified routine must be one declared at the outer level (that is, it must not be internal to another routine).

routine_type. This argument supplies the name of a procedure type or function type.

argument-list. This optional argument list must be a valid argument list for the routine being invoked, as described by routine_type. Note that a pointer obtained with the ADDRESS function can be passed as the argument for a procedural parameter.

For example:

```
MODULE mod1;

FUNCTION operation(operand : REAL) : REAL;
FUNCTION_TYPE;
```

```
FUNCTION square OF TYPE operation;
   BEGIN
   square : = operand * operand;
   END;
FUNCTION cube OF TYPE operation;
   BEGIN
   cube : = operand * operand * operand;
   END;
PROGRAM test;
   VAR res1,res2 : REAL;
   p : ↑ANYTYPE;

   BEGIN
   { this call accomplishes res2 : = square(res1); }
   p : = ADDRESS(square);
   res2 : = INVOKE(p, operation, res1);

   { this call accomplishes res2 : = cube(res1); }
   p : = ADDRESS(cube);
   res2 : = INVOKE(p, operation, res1);
   END;
END;
```

Chapter 10
Queues

Queues are ordered, circular, doubly linked data lists. Each item on the queue is referred to as an *entry*. Entries on a queue are ordered in that there is a distinctive path from each entry to the next. The path is constructed as entries are inserted onto the queue. The queue is circular in that following the path forward from a given entry will lead back to the same entry. The queue is doubly linked because it is possible to travel forward, as well as backward, on the path from entry to entry.

This chapter discusses queue declarations and queue procedures, including examples of typical uses of queues in VAXELN Pascal, followed by a discussion of using queues in interprocess communication.

Queue Declarations

All queue entries are declared using the data type QUEUE_ENTRY.

QUEUE_ENTRY Data Type

QUEUE_ENTRY is a predeclared record type. Each queue entry contains links, or addresses, for two other entries: the one "in front" of it and the one "behind" it. These addresses are stored in a record containing the two addresses:

```
TYPE QUEUE_ENTRY = RECORD
   flink : ↑QUEUE_ENTRY;
```

```
    blink : ↑QUEUE_ENTRY;
    END;
```

The forward link flink is the address of the succeeding queue entry. The backward link blink is the address of preceding queue entry.

In VAXELN Pascal, you can create new queues, insert entries, and remove entries. New entries can be inserted "ahead" of an entry already on the queue, or "after" the entry. Similarly, entries can be removed from any point on the queue.

Larger data structures can be placed on the queue by imbedding the QUEUE_ENTRY structure as the first member of a larger record or structure. It is important to make the queue entry the first member, so that the addresses contained in the queue entry can be used to address the entire structure. For example:

```
VAR myentry: RECORD
    myentryq: QUEUE_ENTRY;
    item : INTEGER;
    END;
```

Since it is possible to traverse from one entry on the queue, around the queue, coming back to that entry, it is usually the practice to designate one entry as the queue *head*. This entry is special in that it is a place holder for the queue list structure and not an array on the queue to contain data.

A queue header specifies the first and last entries on the queue. Because the header contains only the two pointers flink and blink, you can declare it as follows:

```
VAR header: QUEUE_ENTRY;
```

The forward link of the header is the address of the queue head. The backward link of the header is the address of the queue *tail*, the last entry on the queue.

A queue is termed *empty* if the head is the only entry on the queue, as shown in Figure 10-1.

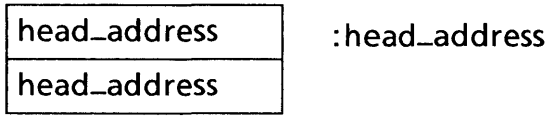


Figure 10-1. An Empty Queue Header

Note that both the forward and backward links in the header address the header itself. A queue entry in this state is considered empty by definition.

Since queues are dynamic structures, it is necessary to initialize or "start" them. In VAXELN Pascal, you use the START_QUEUE procedure to set a queue header to reflect an empty queue. You then use the INSERT_ENTRY procedure to insert an entry onto the queue.

A queue with a single element on it is shown in Figure 10-2.

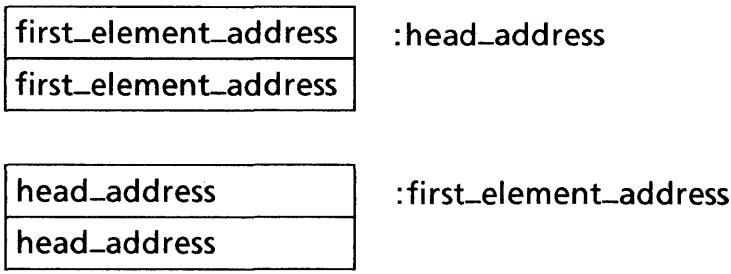


Figure 10-2. A Single-Element Queue

A queue with two elements is shown in Figure 10-3.

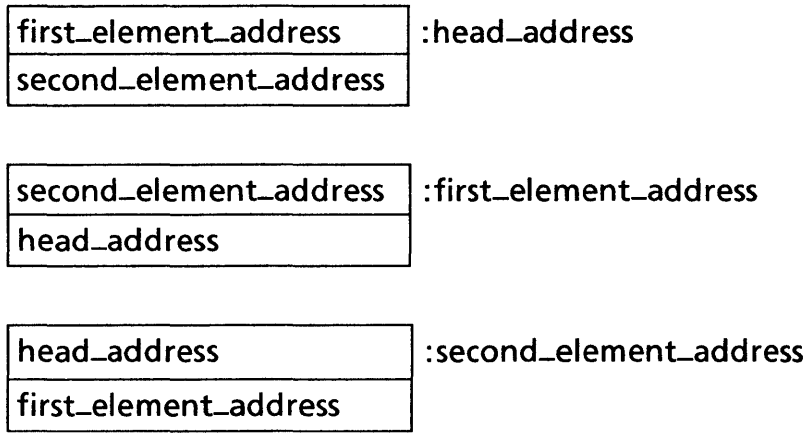


Figure 10-3. A Two-Element Queue

Note that a queue with two elements has a distinct ordering or position for each element. The first element on the queue (going forward from the header) is first_element, the second is second_element. By definition, first_element is at the head of the queue, since it is first. Also by definition, second_element is at the tail of the queue, since it is last.

Entries on the queue can be inserted and removed from arbitrary positions along the queue. For instance, in a first-in-first-out (FIFO) queue, entries are inserted at the tail and removed from the head, thus preserving the removal order as the insertion order. By contrast, in a last-in-first-out (LIFO) queue, entries are inserted at the head and removed from the head; this is also called a *push down* queue.

QUEUE_POSITION Data Type

In VAXELN Pascal, the positions along a queue are represented by the data type QUEUE_POSITION, which is a predeclared enumerated type:

```
TYPE QUEUE_POSITION =
    (QUEUE$HEAD, QUEUE$TAIL, QUEUE$CURRENT);
```

QUEUE_POSITION is used with INSERT_ENTRY or REMOVE_ENTRY to specify the position at which insertion or removal should take place. Note that QUEUE$CURRENT is used only with the REMOVE_ENTRY procedure, to remove the entry at the current position while "walking" the queue (see "Queue Examples," later in this chapter).

Queue Procedures

In creating a new queue, the first steps are declaring a suitable variable as the header and initializing it with the START_QUEUE procedure. Entries are inserted and removed with the procedures INSERT_ENTRY and REMOVE_ENTRY, respectively. Table 10-1 summarizes these procedures.

Table 10-1. Queue Procedures

Procedure	Purpose
INSERT_ENTRY	inserts an entry onto a queue.
REMOVE_ENTRY	removes an entry from a queue.
START_QUEUE	initializes a queue header.

INSERT_ENTRY

The INSERT_ENTRY procedure inserts an entry onto a queue identified by a header, at a position (head or tail) specified with type QUEUE_POSITION. The procedure also informs you if the inserted entry was the first entry (not including the header).

Call Format

```
INSERT_ENTRY(
    header,
    entry,
    first_element,
    position
    )
```

Arguments

header. This argument is a variable of type QUEUE_ENTRY, representing the queue header. The forward link of this entry points to the first queue entry (the one at the head), and the backward link points to the last entry.

entry. This argument is also a variable of type QUEUE_ENTRY, representing the entry to be inserted.

first_element. This argument is a variable of type BOOLEAN. After the INSERT_ENTRY call, the variable's value is TRUE if the queue was empty (except for the header) before the call; otherwise, FALSE.

position. This argument supplies the location of the insertion. The value must be the enumerated constant QUEUE$HEAD or QUEUE$TAIL, where

QUEUE$HEAD specifies that the new entry is inserted at the head.

For example:

```
VAR header: QUEUE_ENTRY;
    firstelement: BOOLEAN;
    myentry: RECORD
      myentryq: QUEUE_ENTRY;
      item: INTEGER;
        { For queue of integers. }
      END;
BEGIN
.
.
START_QUEUE(header);
   { Initialize header. }
   myentry.item : = 1;
   { Define item. }
   INSERT_ENTRY(
     header,
     myentry.myentryq,
     firstelement,
     queue$head
     );
     { Insert entry at head. }
.
.
END.
```

After the insertion, the BOOLEAN variable firstelement is TRUE if the inserted entry was the first.

The position QUEUE$HEAD is merely the position identified by header.flink, and QUEUE$TAIL is the position header.blink.

The position QUEUE$CURRENT cannot be used with INSERT_ENTRY. Because INSERT_ENTRY's first argument is any QUEUE_ENTRY variable, however, you can actually insert entries at any position in a queue. For example, suppose myentry has been inserted in the queue, as above, and is followed by several more entries. You can insert a new entry, mynext, immediately after myentry as follows:

```
VAR header: QUEUE_ENTRY;
    firstelement: BOOLEAN;
    mynext,myentry: RECORD
      entryq: QUEUE_ENTRY;
      item: INTEGER;
      { For queue of integers. }
      END;
BEGIN
.
.  { Insert myentry and various others. }
.
INSERT_ENTRY(
   myentry.entryq,
   { Pretend this is the header.}
   mynext.entryq,
   { Entry to be inserted. }
   firstelement,
   QUEUE$HEAD
   );
   { Insert mynext after myentry. }
.
.
END.
```

Here, myentry.entryq is used as if it were the header. Thus, the entry mynext.entryq is inserted at myentry.entryq.flink.

REMOVE_ENTRY

The REMOVE_ENTRY procedure removes an entry from a queue identified by a header, from the position specified with type QUEUE_POSITION. The procedure also informs you if the removed entry was the last entry (not including the header).

Call Format

```
REMOVE_ENTRY(
    header,
    entry,
    empty,
    position
    )
```

Arguments

header. This argument is a value of type QUEUE_ENTRY identifying the queue from which to remove the entry. (When the position is QUEUE$CURRENT, this argument is a QUEUE_ENTRY value identifying the entry to remove.)

entry. This argument is a variable of type ↑QUEUE_ENTRY that receives a pointer to the QUEUE_ENTRY part of the removed entry. If the queue is empty before the operation, the variable receives the value NIL.

empty. This argument is a BOOLEAN variable that receives the value TRUE if the queue is empty after the removal. Otherwise, it receives the value FALSE.

position. This argument is an enumerated constant specifying the position for removal. The valid values

are QUEUE$HEAD, QUEUE$TAIL, and QUEUE$CURRENT.

For example:

```
VAR header: QUEUE_ENTRY;
    empty: BOOLEAN;
    entryptr: ↑QUEUE_ENTRY;
BEGIN
.
.
  REMOVE_ENTRY(
    header,
    entryptr,
    empty,
    QUEUE$HEAD
    );
  { Remove the entry at the head.}
.
.
END.
```

After this operation, the entry previously at the head is removed from the queue and is identified by entryptr. The pointers in the remaining entries are updated by REMOVE_ENTRY. If the only remaining entry is the header, the variable empty is TRUE and if the queue was already empty, entryptr is NIL.

Note that entryptr does not necessarily point to the record or other variable containing the QUEUE_ENTRY data item. To ensure that it does, we recommend that you model a queue entry with a record as shown above (that is, with the first field having type QUEUE_ENTRY). The pointer returned by REMOVE_ENTRY will then have the address of the record, and it can be typecast to point to the record's

type. (For more information about typecasting, see Chapter 5, "Variables.")

An entry at any position can be removed with QUEUE$CURRENT. In this case, the pointer to the current entry is supplied as the header argument, and QUEUE$CURRENT as the position.

START_QUEUE

The START_QUEUE procedure initializes a queue header for use by INSERT_ENTRY. A queue initialized with START_QUEUE is termed *empty*.

Call Format

```
START_QUEUE(header)
```

Arguments

header. The argument supplies a variable of type QUEUE_ENTRY. The procedure makes the header's forward link and backward link pointers point to the header itself.

Queue Examples

The examples illustrated in this section suggest typical uses of queues in VAXELN Pascal.

Inserting at Tail, Removing from Head

This example is a simplified version of a common technique: inserting items at the queue tail and removing them from the head. In this case, three queue entries are inserted, linking three integers; then they are removed from the queue in the same order:

```
PROGRAM qwalk(OUTPUT);

TYPE
   linkedentry = RECORD
     entryq: QUEUE_ENTRY;
     item: INTEGER;
     END;

VAR
   header: QUEUE_ENTRY;
   entryptr: ↑linkedentry;
   entry: linkedentry;
   entry2: linkedentry;
   entry3: linkedentry;
   firstel,empty: BOOLEAN;

BEGIN
   START_QUEUE(header);
   entry.item : = 1;
   INSERT_ENTRY(
     header,
     entry.entryq,
     firstel,
     QUEUE$TAIL
     );
   entry2.item : = 2;
```

```
    INSERT_ENTRY(
      header,
      entry2.entryq,
      firstel,
      QUEUE$TAIL
      );
    entry3.item : = 3;
    INSERT_ENTRY(
      header,
      entry3.entryq,
      firstel,
      QUEUE$TAIL
      );
    REPEAT
      REMOVE_ENTRY(
        header,
        entryptr:: ↑ QUEUE_ENTRY,
        empty,
        QUEUE$HEAD
        );
      WRITELN('entry item: ',
        entryptr ↑ .item);
    UNTIL empty;
END.
```

"Walking" a Queue

This sequence moves forward in a queue, accessing one entry at a time, beginning with the one at the head (in such a case, you can operate on the linked data items without removing their associated queue entries):

```
TYPE
  entrytype = RECORD
    entryq: QUEUE_ENTRY;
    item: INTEGER;
    END;
```

```
VAR
   header: QUEUE_ENTRY;
   qptr: ↑QUEUE_ENTRY;
   entry1, entry2,... : entrytype;
   { Declare entries. }
BEGIN
   .
   .   { Initialize header, insert entries. }
   .
   .   { Walk the queue: }
   qptr : = header.flink;
   { Identify head entry. }
   REPEAT
     .
     .
   WITH qptr:: ↑entrytype↑ DO BEGIN
     { With the above reference to the record,
     operate as you like on the entryq or item fields;
     for example: }
     WRITELN('Entry's item: ',item);
     .
     .
     END; { End of WITH. }
   qptr : = qptr↑.flink
   { Identify the next entry. }
   UNTIL qptr = ADDRESS(header)
   { End of walk. }
END.
```

Here, the first operation in the REPEAT loop affects the entry at the head. After the operation, qptr is given the location of the next entry. The loop stops when qptr identifies the header, meaning that the last operation affected the tail entry.

The pointer variable qptr (type ↑QUEUE_ENTRY) must be cast to type ↑entrytype (a pointer to a record) and then followed by the indirection character; this way, the WITH statement establishes a reference to the record containing the data item.

You can walk backward in a queue as well, by starting at header.blink (the tail), updating qptr with qptr↑.blink, and using the same UNTIL expression.

Note also that you need not start at the head or tail.

Removing All the Entries from a Queue

This sequence removes all entries from a nonempty queue, starting with the first:

```
VAR
   header: QUEUE_ENTRY;
   qptr: ↑QUEUE_ENTRY;
   empty: BOOLEAN;
   entry: RECORD
     entryq: QUEUE_ENTRY;
     item: integer;
     END;
BEGIN
   REPEAT
     REMOVE_ENTRY(
       header,
       qptr,
       empty,
       QUEUE$HEAD
       );
     { Remove head entry. }
     IF (qptr <>NIL ) THEN BEGIN
       .
       .
     { Entry removed. Do what you like with qptr. }
```

```
        .
        .
      END; { End of IF. }
    UNTIL empty
      { When the last entry is removed, qptr is NIL and
      empty is TRUE. }

    .
    .
END.
```

Walking a Queue and Removing One Entry

This sequence removes a particular entry from a nonempty queue:

```
VAR
  header: QUEUE_ENTRY;
  entryptr,qptr: ↑QUEUE_ENTRY : = NIL;
  empty: BOOLEAN;
  entry: RECORD
    entryq: QUEUE_ENTRY;
    item: integer;
    END;
BEGIN
  qptr : = header.flink;
  { Identify head entry. }
  REPEAT
    IF ...
    { If item is the one you want. }
    THEN BEGIN
      { Remove this entry. }
      REMOVE_ENTRY(
        qptr↑,     { Current entry. }
        entryptr,  { Address of removed entry. }
        empty,
        QUEUE$CURRENT
        );
```

```
        qptr : = ADDRESS(header);
        { Stop walk. }
      END
      ELSE qptr : = qptr ↑ .flink;
      { Identify next entry. }
    UNTIL qptr = ADDRESS(header);
    IF entryptr = NIL
      THEN WRITELN('Entry not found.');
    ELSE ...
    { Do what you like with the entry identified by
    entryptr. }
END.
```

Using Queues in Interprocess Communication

One of the most important uses for queues in VAXELN is their use as a very efficient intra-job, interprocess communication mechanism. They can be used to pass data "messages" between two or more processes within a single job. Using queues this way is more efficient within a single job than using the SEND/RECEIVE procedures. Although the SEND/RECEIVE procedures can be used to send messages between processes in the same job, they are better suited to passing messages between jobs on the same or different systems in a network (see Chapter 12, "Interjob Communication").

Queues can be used efficiently because the INSERT_ENTRY and REMOVE_ENTRY procedures are implemented using the VAX INSQUE/REMQUE instructions. These instructions are non-interruptible and, therefore, provide implicit synchronization of insertion and removal of queue entries. This means that two processes can simultaneously access a queue, one inserting entries and the other removing entries, without additional synchronization.

A semaphore is typically used with each queue to signal the transition of the queue from an empty state to a nonempty state. The INSERT_ENTRY procedure and the INSQUE instruction provide this indication via a self-synchronizing method. Therefore, the queue and the semaphore can work together to provide for both list maintenance and process synchronization and scheduling.

Interprocess Communication Example

The following example module shows how to use queues with semaphores for the most efficient method of interprocess communication in VAXELN. The module is composed of the module data, one procedure and two process blocks. The module data consists of the two queues (a list of free entries and a list of done entries) and two semaphores, one per queue. The procedure initialize initializes the queue heads and the semaphores and fills the free queue with entries.

The first process block, producer, is the producer of data and "done" queue entries. It first allocates a free queue entry (waiting if there are none), calls an imported procedure to "get some data," and then inserts the queue entry into the done queue. If the entry in the done queue is the first one, it signals the semaphore, which awakens the other process.

The second process block, consumer, is the consumer of data and "done" queue entries. It first allocates a done queue entry (waiting until one is produced by the producer process), calls an imported procedure to "use some data," and then inserts the used queue entry into the free list. If the entry in the free queue is the first one, it signals the semaphore, which awakens the other process.

```
MODULE producer_consumer;
IMPORT
  data, get_some_data, use_some_data;
TYPE
  list_entry = RECORD
    q: QUEUE_ENTRY;
    d: data;
    END;
VAR
  free_list, done_list: QUEUE_ENTRY;
  free_non_empty, done_non_empty: SEMAPHORE;
PROCEDURE initialize;
  VAR
    p: ↑list_entry;
    empty: BOOLEAN;
    free_count: INTEGER;
  BEGIN
    START_QUEUE(free_list);
    START_QUEUE(done_list);
    CREATE_SEMAPHORE(free_non_empty, 1, 1);
    CREATE_SEMAPHORE(done_non_empty, 0, 1);
    FOR free_count : = 1 TO 10 DO
      BEGIN
        NEW(p);
        INSERT_ENTRY(
          free_list,
          p:: ↑QUEUE_ENTRY↑,
          empty,
          QUEUE$TAIL
          );
      END;
  END;
```

```
PROCESS_BLOCK producer;
  VAR
    p: ↑list_entry;
    first, empty: BOOLEAN;
    stat: INTEGER;
  BEGIN
    REPEAT
      REPEAT
        REMOVE_ENTRY(
          free_list,
          p:: ↑QUEUE_ENTRY,
          empty,
          QUEUE$HEAD
          );
        IF p = NIL THEN
          WAIT_ANY(free_non_empty);
      UNTIL p<> NIL;
      get_some_data(p↑.d);
      INSERT_ENTRY(
        done_list,
        p:: ↑QUEUE_ENTRY↑,
        first,
        QUEUE$TAIL
        );
      IF first THEN
        SIGNAL(done_non_empty, STATUS : = stat);
    UNTIL FALSE;
  END;
```

```
PROCESS_BLOCK consumer;
  VAR
    p: ↑list_entry;
    first, empty: BOOLEAN;
    stat: INTEGER;
  BEGIN
    REPEAT
      REPEAT
        REMOVE_ENTRY(
          done_list,
          p:: ↑QUEUE_ENTRY,
          empty,
          QUEUE$HEAD
          );
        IF p = NIL THEN
          WAIT_ANY(done_non_empty);
      UNTIL p<> NIL;
      use_some_data(p ↑ .d);
      INSERT_ENTRY(
        free_list,
        p:: ↑QUEUE_ENTRY ↑ ,
        first,
        QUEUE$TAIL
        );
      IF first THEN
        SIGNAL(free_non_empty, STATUS : = stat);
    UNTIL FALSE;
  END;
END;
```

Chapter 11
Subprocesses and Synchronization

Introduction

The VAXELN kernel controls the sharing of system resources and synchronizes communication among the various programs in the system. The kernel maintains all information about the system data and about the user programs defined for a particular system.

The kernel provides most of its services through the set of structured variables called kernel objects and the procedures to manipulate them, called kernel services. These services create, delete, or otherwise affect the state of the kernel objects represented by the system data types PROCESS, AREA, EVENT, SEMAPHORE, MESSAGE, PORT, NAME, and DEVICE. (For a detailed discussion of the VAXELN kernel and the kernel objects, see the *VAXELN User's Guide.*)

As described in Chapter 2, "Program Structure," a PROGRAM block is the main routine of a job's master process. A *subprocess* of a job is created by a call to the CREATE_PROCESS kernel service. One argument of the CREATE_PROCESS call is the name of a process block, and this process block is the main routine of the subprocess.

All subprocesses of a job execute in parallel with the master process and with each other. In general, subprocesses must synchronize their activities using

the special data types and routines provided in VAXELN Pascal for this purpose.

This chapter discusses process blocks and the kernel services relating to processes and synchronization. A brief description of each procedure is given, followed by the VAXELN Pascal call format, detailed argument descriptions, and status values.

The relationship between the kernel services described in this chapter and those described in Chapter 12, "Interjob Communication," is as follows:

- The DELETE procedure (defined in this chapter) may be used to delete a MESSAGE or AREA object, as well as any of the objects related to processes and synchronization.
- The SIGNAL procedure (defined in this chapter) may also be used to signal an AREA object.
- The WAIT_ANY and WAIT_ALL procedures (defined in this chapter) may also be used to wait for the arrival of a message or to wait for a signal to an AREA object.
- Messages and ports (defined in the next chater) may be used for communication between processes in the same job, as well as between different jobs.

In addition, this chapter discusses process UICs and the authorization procedures, the Authorization Service utility, program loader utility, and exit utility procedures, the MUTEX data type, and the VAXELN procedures that perform operations on mutexes.

Process Blocks

The syntax for a process block declaration is shown in Figure 11-1.

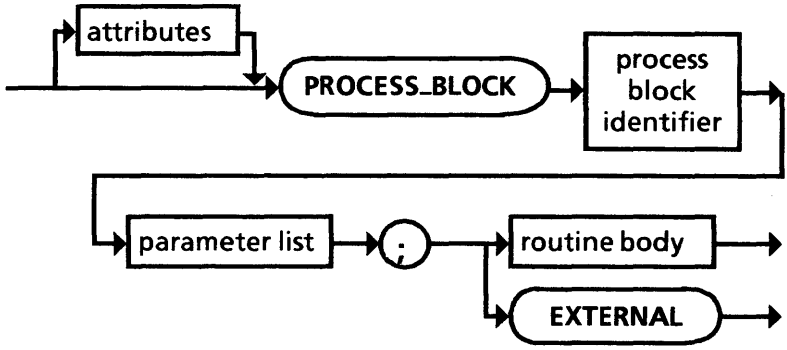


Figure 11-1. Process Block Declaration Syntax

The process block identifier is declared at the outer level as the name of the process block. This declaration is only allowed at the outer level.

The routine body gives the local declarations and executable code for the process block. One of the attributes UNDERFLOW and NOUNDERFLOW may be specified. The attribute applies to the routine body, as explained in Chapter 2, "Program Structure."

The EXTERNAL directive may be used to indicate that the routine body is coded in another programming language. In this case, neither UNDERFLOW nor NOUNDERFLOW is allowed.

The optional parameter list specifies a list of process parameters, arguments for which can be supplied by CREATE_PROCESS calls. The arguments in CREATE_PROCESS match these parameters positionally, from left to right. The parameter list has the same format as for procedures (for details, see Chapter 8, "Procedures and Functions"). If supplied, the

parameters must meet the following constraints, which are related to the way arguments are passed to the process block (via a call to CREATE_PROCESS, the call being made in another process):

- The total number of parameters (or arguments) must not exceed 31, including any conformant extents.
- The parameters must be value or VAR parameters (not procedural parameters).
- The data types of value parameters are restricted to ordinal types, set types with ORD(maxelement) < 32, pointer types, and types REAL, PROCESS, AREA, EVENT, SEMAPHORE, MESSAGE, NAME, and DEVICE. (These types all have values expressible in 32 bits.) Types with the BIT attribute are not allowed, and the REFERENCE attribute cannot be used.

Subprocess Activation and Termination

A subprocess is created when another process in the job calls CREATE_PROCESS. The kernel establishes a new stack (P1 virtual address space) for the new process and prepares it for execution, beginning at the first statement within the process block's compound statement (BEGIN ... END). The new process is in the Ready state; it will begin actual execution now or later, depending on its priority.

A subprocess terminates when any one of the following occurs:

- Execution of its compound statement (BEGIN ... END) terminates.
- The process calls the EXIT procedure.
- The process is deleted by a call to DELETE.

- An unhandled exception occurs in the process; this may be an unhandled QUIT signaled by another process.
- The job's master process terminates.

When a subprocess terminates, its stack (P1 address space) is returned to the kernel. In addition, the kernel takes action so that:

- In any process of the job that is currently waiting for termination of this process, the wait is satisfied.
- If the call to CREATE_PROCESS that activated this process specified an exit_status argument, the terminated process's exit status is stored in the designated data item. (It's an unpredictable error if the data item has been freed since the call to CREATE_PROCESS.)

These actions are not taken in the case where the subprocess is terminated because the master process terminates.

VAXELN provides utility procedures that can be used to establish an exit handler to perform cleanup operations following the termination of a job with the EXIT procedure. These procedures are described under "Exit Utility Procedures," later in this chapter.

Calling Conventions for Process Blocks

The argument list for a process block is created by the kernel using the process block arguments specified in the call to CREATE_PROCESS. The conventions for this argument list are the same as for a procedure's or function's argument list, except that descriptors are not used. Instead, the conformant extents are in the argument list, immediately following the conformant argument that defines them. (For more information, see "Conformant Parameters," in Chapter 8.)

Kernel Services for Processes and Synchronization

The kernel services described in this section relate to creating, terminating, and scheduling processes and jobs, and to the synchronization of processes, events, semaphores, ports, devices, and areas. Table 11-1 summarizes these procedures alphabetically.

Table 11-1. Kernel Services for Processes and Synchronization

Procedure	Purpose
CLEAR_EVENT	sets the state of an EVENT value to cleared.
CREATE_EVENT	creates and initializes an EVENT value.
CREATE_JOB	creates a new job.
CREATE_PROCESS	creates a new process executing a specified process block.
CREATE_SEMAPHORE	creates and initializes a semaphore.
CURRENT_PROCESS	returns a variable identifying the process from which it is called.
DELETE	removes a kernel object from the system.
DISABLE_SWITCH	disables process rescheduling in current job.
ENABLE_SWITCH	resumes process rescheduling in current job.

Table 11-1. Continued

Procedure	Purpose
EXIT	ends the current process.
INITIALIZATION_DONE	informs the kernel that the current job has completed the initialization sequence.
RESUME	resumes a suspended process.
SET_JOB_PRIORITY	sets scheduling priority of the current job.
SET_PROCESS_PRIORITY	sets scheduling priority of the specified process.
SIGNAL	signals a process, semaphore, event, or area.
SUSPEND	suspends execution of a process.
WAIT_ALL	makes the calling process wait for **all** object values to satisfy the wait.
WAIT_ANY	makes the calling process wait for **any** object value to satisfy the wait.

CLEAR_EVENT

The CLEAR_EVENT procedure sets the state of an EVENT object to EVENT$CLEARED.

Call Format

```
CLEAR_EVENT(
   event,
   STATUS : = stat
   )
```

Arguments

event. This argument supplies the EVENT value identifying the EVENT object to be cleared.

stat. This optional argument is an INTEGER variable that receives the completion status of CLEAR_EVENT.

Status Values

KER$_SUCCESS. The procedure completed successfully.

KER$_BAD_TYPE. The event argument is not of type EVENT.

KER$_BAD_VALUE. The event argument is invalid or refers to a deleted event.

CREATE_EVENT

The CREATE_EVENT procedure creates and initializes an EVENT object.

Call Format

```
CREATE_EVENT(
   event,
   initial_state,
   STATUS: = stat
   )
```

Arguments

event. This argument is a variable of type EVENT that receives the identifier of the new event.

initial_state. This argument supplies a value of the predeclared enumerated type EVENT_STATE:

```
TYPE EVENT_STATE = (EVENT$CLEARED,
EVENT$SIGNALED)
```

The value gives the initial state of the EVENT object.

stat. This optional argument is an INTEGER variable that receives the completion status of CREATE_EVENT.

Status Values

KER$_SUCCESS. The procedure completed successfully.

KER$_NO_OBJECT. No free job object table entries are currently available. There are a maximum of 1024 object table entries per job.

KER$_NO_POOL. No free system pool is currently available. The size of the system pool can be set by the System Builder utility.

CREATE_JOB

The CREATE_JOB procedure creates a new job, which executes a specified program image. Note that CREATE_JOB runs a program image already in the system (via the System Builder or via the LOAD_PROGRAM procedure); it cannot add a new image to a system.

Call Format

```
CREATE_JOB(
   job_port,
   program,
   argument-list,
   NOTIFY : = exit_port,
   STATUS : = stat
   )
```

Arguments

job_port. This argument is a PORT variable that receives the new job port value. The value can be used by the caller of CREATE_JOB to send messages to the new job. The same value is returned within the new job by the JOB_PORT procedure.

program. This argument is a string that supplies the name of the program the job is to run. The name is one of the programs specified to the System Builder or loaded via the LOAD_PROGRAM procedure (see "Program Loader Utility Procedures," later in this chapter).

argument-list. This is an optional list of strings supplied as arguments to the program. Arguments can also be supplied to the program with the System Builder, as part of a program description. Note that any

arguments supplied here override arguments supplied with the System Builder.

exit_port. This optional argument supplies a PORT value for termination notification. If this argument is present, a "termination message" is sent to the port when the new job terminates. (Note that the port must already be created.) The message data of the termination message is the INTEGER value making up the completion status of the created job's master process. The job's master process can return an explicit status with the EXIT procedure; if it specifies no status and completes successfully, the default status returned in the termination message is 1 (success). If the argument is omitted, no message is sent.

stat. This optional argument is an INTEGER variable that receives the completion status of CREATE_JOB.

Status Values

KER$_SUCCESS. The procedure completed successfully.

KER$_BAD_LENGTH. A string argument was too long.

KER$_NO_ACCESS. An argument specified is not accessible to the calling program.

KER$_NO_MEMORY. No free pages of physical memory are currently available.

KER$_NO_PAGE_TABLE. No free process page table is currently available. The number of process page tables can be set by the System Builder utility.

KER$_NO_POOL. No free system pool is currently available. The size of the system pool can be set by the System Builder utility.

KER$_NO_PORT. No free system port table entries are currently available. The size of the system port table can be set by the System Builder utility.

KER$_NO_SUCH_PROGRAM. No program with the specified name can be found in the program list created by the System Builder utility.

CREATE_PROCESS

The CREATE_PROCESS procedure creates a new process executing a specified process block.

Call Format

```
CREATE_PROCESS(
    process,
    subprocess_name,
    argument-list,
    EXIT : = exit_status,
    STATUS : = stat
    )
```

Arguments

process. This argument is a PROCESS variable that receives the identifier of the new process.

subprocess_name. This argument supplies the name of the process block to run.

argument-list. This is a list of zero to 31 arguments, separated by commas. The arguments are passed to the corresponding parameters in the process block's parameter list, just as in a procedure or function invocation. Note, however, that only the positional argument notation (as explained under "Call Format Conventions" in Chapter 1) can be used; the nonpositional notation cannot be used.

exit_status. This optional argument is an INTEGER variable that receives the final status of the created process. Such a value can be returned with the EXIT

procedure. By convention, odd-numbered values indicate success; even-numbered values indicate errors (not necessarily fatal). If the argument is omitted, no status is returned.

stat. This optional argument is an INTEGER variable that receives the completion status of CREATE_PROCESS.

As part of its internal operation, the call to CREATE_PROCESS passes the following addresses to the new process:

- The address of an argument passed to a VAR parameter
- A pointer value passed to a value parameter
- The address of the exit status item

It is an error if any of these addresses denote an item in the creating process's P1 address space (for example, a local variable). The compiler and the kernel detect some violations of this rule. Undetected violations lead to unpredictable behavior, since P1 address values from the creating process have no meaning in the new process.

Status Values

KER$_SUCCESS. The procedure completed successfully.

KER$_BAD_COUNT. The procedure call specified an incorrect number of arguments.

KER$_BAD_VALUE. The exit status variable is in P1 space.

KER$_NO_ACCESS. An argument specified is not accessible to the calling program.

KER$_NO_OBJECT. No free job object table entries are currently available. There are a maximum of 1024 object table entries per job.

KER$_NO_MEMORY. No free pages of physical memory are currently available.

KER$_NO_PAGE_TABLE. No free process page table is currently available. The number of process page tables can be set by the System Builder utility.

KER$_NO_POOL. No free system pool is currently available. The size of the system pool can be set by the System Builder utility.

KER$_NO_STATUS. The process was deleted; therefore, no exit status value is available to return. (This value is returned only as an exit status, in the CREATE_PROCESS exit_status argument.)

CREATE_SEMAPHORE

The CREATE_SEMAPHORE procedure creates and initializes a semaphore.

Call Format

```
CREATE_SEMAPHORE(
   semaphore,
   initial_count,
   maximum_count,
   STATUS : = stat
   )
```

Arguments

semaphore. This argument is a SEMAPHORE variable that receives the identifier of the new semaphore.

initial_count. This argument is an INTEGER expression that supplies the initial semaphore count. The initial count must not exceed the maximum count.

maximum_count. This argument is an INTEGER expression that supplies the maximum semaphore count. Signaling the semaphore beyond this count is an error.

stat. This optional argument is an INTEGER variable that receives the completion status of CREATE_SEMAPHORE.

Status Values

KER$_SUCCESS. The procedure completed successfully.

KER$_BAD_VALUE. The specified initial count is greater than the maximum count.

KER$_NO_OBJECT. No free job object table entries are currently available. There are a maximum of 1024 object table entries per job.

KER$_NO_POOL. No free system pool is currently available. The size of the system pool can be set by the System Builder utility.

CURRENT_PROCESS

The CURRENT_PROCESS procedure returns the PROCESS value identifying the process from which it is called.

Call Format

```
CURRENT_PROCESS(
   process,
   STATUS : = stat
   )
```

Arguments

process. This argument is a PROCESS variable that receives the identifier of the calling process.

stat. This optional argument is an INTEGER variable that receives the completion status of CURRENT_PROCESS. KER$_SUCCESS is the only possible status.

DELETE

The DELETE procedure removes a PROCESS, AREA, EVENT, SEMAPHORE, MESSAGE, PORT, NAME, or DEVICE object from the system.

Call Format

```
DELETE(
   value,
   STATUS: = stat
   )
```

Arguments

value. This argument supplies a value of type PROCESS, AREA, EVENT, SEMAPHORE, MESSAGE, PORT, NAME, or DEVICE.

stat. This optional argument is an INTEGER variable that receives the completion status of DELETE.

Notes

The result of deleting the various objects is as follows:

PROCESS Objects

When a process is deleted, if any other process is waiting for its termination, that aspect of its wait

condition is satisfied permanently. When a master process is deleted, all subprocesses in the same job are also deleted, along with all data and kernel objects created by any processes in the job. The exit status of a deleted process is KER$_NO_STATUS.

AREA Objects

An area can be deleted by any process of a job that has created or mapped the area. The memory associated with the area is deleted when the last referencer deletes its reference.

EVENT and SEMAPHORE Objects

When an event or semaphore is deleted, any waiting processes are removed from their wait states immediately; the status of WAIT_ANY or WAIT_ALL is KER$_BAD_VALUE.

MESSAGE Objects

When a MESSAGE object is deleted, the message is unavailable for sending or receiving, and any pointers to the message's data buffer become invalid.

PORT Objects

When a port is deleted, any connected port (when the deleted port is in a circuit) is disconnected, any messages at the port are deleted, and the wait conditions of any waiting processes are satisfied with the completion status KER$_BAD_VALUE.

NAME Objects

When a universal name is deleted, the Network Service on each node ensures that the deletion is reflected in the list of universal names. The deletion of local names

is performed by the kernel on the local node and does not involve the Network Service.

DEVICE Objects

When a DEVICE object is deleted, the memory used for its communication region is deleted, and any pointers to that memory become invalid. The interrupt service routine is disconnected from the interrupt vector. Any waiting processes are removed from their wait states immediately, with the completion status KER$_BAD_VALUE.

Status Values

KER$_SUCCESS. The procedure completed successfully.

KER$_BAD_STATE. A device specified to DELETE has an interrupt pending.

KER$_BAD_VALUE. The value argument is invalid or refers to an object that was deleted.

KER$_NO_ACCESS. An argument specified is not accessible to the calling program.

DISABLE_SWITCH

The DISABLE_SWITCH procedure disables process switching for the job from which it is called. The calling process continues executing, regardless of the priorities of other processes in the job, until switching is reenabled with ENABLE_SWITCH.

Note: Process switching is reenabled automatically if the process calls EXIT or deletes itself.

Call Format

```
DISABLE_SWITCH(STATUS : = stat)
```

Arguments

stat. This optional argument is an INTEGER variable that receives the completion status of DISABLE_SWITCH.

Status Values

KER$_SUCCESS. The procedure completed successfully.

KER$_COUNT_OVERFLOW. The DISABLE_SWITCH procedure was called more times than ENABLE_SWITCH.

ENABLE_SWITCH

The ENABLE_SWITCH procedure restores preemptive process scheduling, or switching, for the calling job. When process switching is enabled, the control of the CPU is given to the highest priority process in the job that is ready to run.

Call Format

```
ENABLE_SWITCH(STATUS: = stat)
```

Arguments

stat. This optional argument is an INTEGER variable that receives the completion status of ENABLE_SWITCH.

Notes

The procedures ENABLE_SWITCH and DISABLE_SWITCH have a feature that allows them to be called with reasonable effects from nested routines. The implementation uses a counter that is incremented whenever DISABLE_SWITCH is called and

decremented whenever ENABLE_SWITCH is called. Switching is enabled only when the number of calls to ENABLE_SWITCH is equal to the number of calls to DISABLE_SWITCH for a given process. For example:

```
PROCEDURE a;
   BEGIN
   DISABLE_SWITCH;
   .
   .
   ENABLE_SWITCH;
   END;
PROCEDURE b;
   BEGIN
   DISABLE_SWITCH;
   .
   .
   a;     { Call procedure a. }
   ENABLE_SWITCH;
   END;
```

Here, procedure b disables process switching and then calls procedure a. Procedure a also disables process switching during its execution and then calls ENABLE_SWITCH. This call does not reenable process switching, however, since that would cause an error upon returning to procedure b. Process switching is reenabled only when procedure b calls ENABLE_SWITCH.

Status Values

KER$_SUCCESS. The procedure completed successfully.

KER$_COUNT_UNDERFLOW. The ENABLE_SWITCH procedure was called more times than DISABLE_SWITCH.

EXIT

The EXIT procedure causes an immediate exit from the calling process. Block activations are not terminated individually; that is, there is no "unwinding" of the stack. If the calling process is the master process, all the objects it owns (including subprocesses) are deleted. All open files are closed.

Call Format

```
EXIT(
   EXIT_STATUS : = exit,
   STATUS : = stat
   )
```

Arguments

exit. This optional argument supplies an INTEGER expression giving the exit status of the current process to its creator. If omitted, the creating process receives a status value indicating that no status was returned.

stat. This optional argument is an INTEGER variable that receives the completion status of EXIT. KER$_SUCCESS is the only possible status.

Notes

If process switching was disabled by the calling process, it is reenabled automatically when the EXIT procedure is called.

VAXELN provides the DECLARE_EXIT_HANDLER procedure to establish an exit handler to perform cleanup operations following the termination of a job with the EXIT procedure (see "Exit Utility Procedures," later in this chapter).

INITIALIZATION_DONE

The INITIALIZATION_DONE procedure informs the kernel that the calling job has completed an initialization sequence, and other programs can be started if specified.

Call Format

```
INITIALIZATION_DONE(STATUS : = stat)
```

Arguments

stat. This optional argument is an INTEGER variable that receives the completion status of INITIALIZATION_DONE.

Status Values

KER$_SUCCESS. The procedure completed successfully.

KER$_NO_INITIALIZATION. The calling program does not have the *Init required* characteristic.

RESUME

The RESUME procedure resumes a suspended process. A resumed process is ready to run, but not necessarily running. If the process was waiting when it was suspended, the wait is repeated when it is resumed, as if the WAIT_ANY or WAIT_ALL procedure were called again. Any asynchronous exceptions that occurred during the suspension are raised before the wait is performed, however, including the exception KER$QUIT_SIGNAL that results from signaling the process itself.

Call Format

```
RESUME(
   process,
   STATUS : = stat
   )
```

Arguments

process. This argument supplies a value of type PROCESS that identifies the process to be resumed.

stat. This optional argument is an INTEGER variable that receives the completion status of RESUME.

Status Values

KER$_SUCCESS. The procedure completed successfully.

KER$_BAD_STATE. A process specified to RESUME is not suspended.

KER$_BAD_TYPE. The first argument is not of type PROCESS.

KER$_BAD_VALUE. The process argument is invalid or identifies a process that no longer exists.

SET_JOB_PRIORITY

The SET_JOB_PRIORITY procedure sets the scheduling priority of the current job.

Call Format

```
SET_JOB_PRIORITY(
   priority,
   STATUS : = stat
   )
```

Arguments

priority. This argument is an integer in the range 0–31 that supplies the new priority. Priority 0 is the highest.

stat. This optional argument is an INTEGER variable that receives the completion status of SET_JOB_PRIORITY.

Status Values

KER$_SUCCESS. The procedure completed successfully.

KER$_BAD_VALUE. The priority argument is out of range.

SET_PROCESS_PRIORITY

The SET_PROCESS_PRIORITY procedure sets the scheduling priority of the specified process.

Call Format

```
SET_PROCESS_PRIORITY(
    process,
    priority,
    STATUS : = stat
    )
```

Arguments

process. This argument supplies the PROCESS value identifying the process whose priority is to be changed.

priority. This argument supplies the new priority as an integer in the range 0–15. Priority 0 is the highest.

stat. This argument is an INTEGER variable that receives the completion status of SET_PROCESS_PRIORITY.

Status Values

KER$_SUCCESS. The procedure completed successfully.

KER$_BAD_TYPE. The first argument is not of type PROCESS.

KER$_BAD_VALUE. Either the process argument is invalid or refers to a deleted process, or the priority argument is out of range.

SIGNAL

The SIGNAL procedure signals a process, area, event, or semaphore.

Call Format

```
SIGNAL(
   value,
   STATUS: = stat
   )
```

Arguments

value. This argument supplies a value of type PROCESS, AREA, EVENT, or SEMAPHORE.

stat. This optional argument is an INTEGER variable that receives the completion status of SIGNAL.

Notes

The result of signaling the various objects is as follows:

PROCESS Objects

A process can be signaled to quit with the SIGNAL procedure. The process must establish a handler for the

exception KER$_QUIT_SIGNAL. If it does not handle the exception, it is forced to exit.

AREA Objects

When a referencing process is finished with its exclusive access to an area, the SIGNAL procedure allows the next waiting process to gain explicit access. It is an error to signal an area if the area is not "locked" by any process.

If the area is of zero length, the object represents a named interjob binary semaphore, in which case the semaphore count is incremented and tested. If the new count is greater than zero, the first waiting process in the semaphore's queue whose wait conditions can be satisfied is continued, and the count is decremented. If no processes are waiting, or if none of the waiting processes can continue, the count is not decremented.

EVENT Objects

SIGNAL sets the state of an event to SIGNALED and continues all waiting processes whose wait conditions can be satisfied.

SEMAPHORE Objects

Signaling a semaphore increments and then tests the semaphore count. If the new count is greater than zero, the first waiting process in the semaphore's queue whose wait conditions can be satisfied is continued, and the count is decremented. If no processes are waiting, or if none of the waiting processes can continue, the count is not decremented. At most, one process continues as a result of signaling a semaphore.

Status Values

KER$_SUCCESS. The procedure completed successfully.

KER$_BAD_TYPE. The value argument identifies an object that cannot be signaled.

KER$_BAD_VALUE. The value argument is invalid or refers to a deleted object.

KER$_COUNT_OVERFLOW. SIGNAL was called for a semaphore already at its maximum count.

SUSPEND

The SUSPEND procedure suspends the execution of a process. If the process is currently waiting, as a result of WAIT_ANY or WAIT_ALL, it is removed immediately from the Waiting state and then suspended. If the process is subsequently resumed, the wait is repeated.

Call Format

```
SUSPEND(
   process,
   STATUS : = stat
   )
```

Arguments

process. This argument supplies a value of type PROCESS identifying the process to be suspended.

stat. This optional argument is an INTEGER variable that receives the completion status of SUSPEND.

Status Values

KER$_SUCCESS. The procedure completed successfully.

KER$_BAD_TYPE. The first argument is not of type PROCESS.

KER$_BAD_VALUE. The process argument is invalid or refers to a deleted process.

WAIT_ALL and WAIT_ANY

The WAIT_ALL and WAIT_ANY procedures are used to make a process wait for one or more objects, including processes, areas, events, semaphores, ports, and devices. WAIT_ANY allows the invoking process to proceed if *any* of the wait conditions is satisfied; WAIT_ALL requires that *all* the conditions be satisfied simultaneously. WAIT_ANY identifies the object that satisfied the wait.

Call Formats

```
WAIT_ALL(
    object-list,
    RESULT : = wait_result,
    TIME : = tvalue,
    STATUS : = stat
    )
WAIT_ANY(
    object-list,
    RESULT : = wait_result,
    TIME : = tvalue,
    STATUS : = stat
    )
```

Arguments

object-list. This list supplies zero to four values of type PROCESS, AREA, EVENT, SEMAPHORE, PORT, or DEVICE, separated by commas. If no values are listed,

the wait is satisfied by the timeout if one is specified, or immediately if none is specified.

wait_result. This optional argument is an INTEGER variable that receives the argument number of the object that satisfied the wait. The value 0 means that the wait was satisfied by a timeout, as specified by the tvalue argument. Otherwise, the value placed in wait_result identifies, for WAIT_ANY, the object that satisified the wait, where 1 indicates the first object in the list, and so forth. With WAIT_ALL, 0 means that the procedure timed out, and otherwise the result is an integer in the range 1–4, the exact value being unpredictable. The value of wait_result is undefined if the procedures terminate unsuccessfully.

tvalue. This optional argument supplies an absolute time or time interval. At the specified absolute time, or after the specified interval, the wait is satisfied regardless of the states of the specified objects.

stat. This optional argument is an INTEGER variable that receives the completion status of WAIT_ALL or WAIT_ANY.

Notes

The WAIT procedures return immediately if one of their objects is deleted. The deletion is indicated by KER$_BAD_VALUE. Both procedures also return immediately if the necessary conditions were satisfied already (before the call). Therefore, the elapsed time is only the time required to perform a procedure call, and any specified timeout value is irrelevant.

A timeout value of zero may be used to ensure that the WAIT procedure returns immediately. If the returned wait_result value is zero, it means the wait condition

specified by the objects was not satisfied. (That is, the objects are tested before the timeout.)

Note that, if more than one wait condition satisifies the wait, the wait_result argument does not have any predictable value.

Both WAIT procedures delay the execution of the invoking process until either the wait condition or timeout is satisfied. For example:

```
PROGRAM driver;

VAR
   ready: EVENT;
   unit_avail: SEMAPHORE;
   stat: INTEGER;
   one_second: LARGE_INTEGER;
   satisifier: INTEGER;

BEGIN
   CREATE_EVENT(ready,EVENT$CLEARED);
   CREATE_SEMAPHORE(unit_avail,1,1);
   one_second : = TIME_VALUE('0000 00:00:01.00');
   .
   .
   { Wait for 1 second, for a process to signal ready,
   or for a process to signal unit_avail, whichever
   comes first. }

   WAIT_ANY(
     ready,
     unit_avail,
     RESULT : = satisfier,
     TIME : = one_second,
     STATUS : = stat
     );
```

```
    CASE satisfier OF
      0: { Timeout.... }
      1: { ready (or both) signaled....}
      2: { unit_avail signaled....}
      END;
  END.
```

WAIT_ANY waits for any one of a number of conditions to occur, up to a specified time. It might be used in a device driver to wait for a device interrupt or device timeout. In a multiport server, it might wait for a message to arrive on any one of several ports.

If an asynchronous exception (such as KER$_POWER_SIGNAL) is delivered to a waiting process, several actions are possible, depending on the action of the exception handler:

- If the handler returns FALSE, the exception is "resignaled," meaning that the stack is searched for another handler; here the process may not reenter the waiting state.
- If the handler simply returns TRUE (meaning "exception handled"), the process reenters the waiting state.
- If the the handler is exited with a GOTO statement, the process may not reenter the waiting state.

The conditions for satisfying waits for the various objects and the effects of waiting for each type of object (both procedures have the same effect on their arguments) are as follows:

PROCESS Objects

A wait for a process is satisfied when it terminates. Waiting for a process causes no modification to the

object, and all waiting processes continue if their wait conditions are otherwise satisfied.

AREA Objects

A wait for an AREA object is satisfied when the object is signaled. Waiting for an area implies that, after the wait is satisfied, the waiting process has exclusive access to the area until a complementary signal is sent. When a referencing job's main process is deleted, a check is made and if the process being deleted is the owner process, the area is implicitly signaled. If the process being deleted is the last referencer, the area is deleted.

If the area is of zero length, the object represents a named interjob binary semaphore, in which case the semaphore count is decremented if the wait is satisfied by signaling the semaphore.

EVENT Objects

A wait for an event is satisfied when the object is signaled. Waiting for an event causes no modification to the object, and all waiting processes continue if their wait conditions are otherwise satisfied.

SEMAPHORE Objects

A wait for a semaphore is satisfied when the object is signaled. Waiting for a semaphore causes the semaphore count to be decremented if the wait is satisfied by signaling the semaphore; at most one process continues as the result of signaling a semaphore.

PORT Objects

A wait for a port (including a port in a circuit) is satisfied when it has a message in it. Waiting for a port causes no modification to the object, and all waiting

processes continue if their wait conditions are otherwise satisfied.

DEVICE Objects

A wait for a DEVICE object is satisfied when the state of the object is "signaled" (the result of the SIGNAL_DEVICE procedure, called from an interrupt service routine). Waiting for a device causes the DEVICE object to be cleared if the wait is satisfied by the DEVICE object. That is, only one process continues as a result of the action of an interrupt service routine.

Status Values

KER$_SUCCESS. The procedure completed successfully.

KER$_BAD_COUNT. The procedure call specified an incorrect number of arguments.

KER$_BAD_TYPE. An argument in the object-list is not a type that can be waited for.

KER$_BAD_VALUE. An argument in the object-list is invalid or refers to a deleted object.

KER$_NO_ACCESS. An argument specified is not accessible to the calling program.

Process UICs

Associated with each process in a VAXELN system is a user name and a user identification code, or *UIC*. The primary use of a process's user name and UIC is to enable the process's requests to remote systems to be authorized by the remote system.

The UIC is an INTEGER value that provides a shorthand way of identifying a user or group of users. The user name and UIC are maintained by the VAXELN Authorization Service and are inherited by a process from the process or job that created it. A process can also set its own user name and UIC to any desired values by calling the SET_USER kernel procedure.

A process can determine its own user name and UIC by calling the GET_USER kernel procedure. Since the VAXELN security features are based upon validating network requests, a process can also determine the user name and UIC of the process from which it has accepted a circuit connection.

For a detailed discussion of VAXELN system security, including the Authorization Service and the use of UICs, refer to the *VAXELN User's Guide*. The procedures provided by the Authorization Service to maintain the authorization database are described under "Authorization Service Utility Procedures," later in this chapter.

Authorization Procedures

The procedures described in this section set or return the user identity of processes. Table 11-2 summarizes these procedures.

Table 11-2. Authorization Procedures

Procedure	Purpose
GET_USER	returns the user identity of a process.
SET_USER	sets the user identity of the current process.

Note: To use these procedures, include the module $KERNEL from the RTLOBJECT library in the compilation of your program.

GET_USER

The GET_USER kernel procedure returns the user identity of either the calling process or the partner process connected by a circuit to the caller's port. To use the procedure, you must include the module $KERNEL in the compilation.

Call Format

```
KER$GET_USER(
    status,
    circuit,
    username,
    uic
    )
```

Arguments

status. This optional argument is an INTEGER variable that receives the completion status of GET_USER.

circuit. This optional argument supplies a PORT value specifying the partner process's port in the circuit. If this argument is supplied, the port must be currently connected in a circuit that the caller has accepted with the ACCEPT_CIRCUIT procedure. Valid information is not returned if the caller initiated the connection with CONNECT_CIRCUIT; that is, GET_USER can only provide information about the object of a connection, not the subject.

username. This optional argument receives a string of up to 20 characters which is the user name of either the calling process or the partner process.

uic. This optional argument is an INTEGER variable that receives the user identification code of either the

calling process or the partner process. If the circuit is from a remote user, but there is no Authorization Service available in the system (that is, the *Authorization required* characteristic on the *Edit Network Node Characteristics* System Builder menu is "No"), GET_USER returns zero for the UIC parameter.

Status Values

KER$_SUCCESS. The procedure completed successfully.

KER$_BAD_LENGTH. The username argument is too long.

KER$_NO_ACCESS. An argument specified is not accessible to the calling program.

SET_USER

The SET_USER kernel procedure sets the user identity of the current process. To use the procedure, you must include the module $KERNEL in the compilation.

Call Format

```
KER$SET_USER(
   status,
   username,
   uic
   )
```

Arguments

status. This optional argument is an INTEGER variable that receives the completion status of SET_USER.

username. This argument supplies a string of up to 20 characters giving the user name to be associated with the process.

uic. This argument is an INTEGER value that supplies the user identification code to be associated with the process.

Status Values

KER$_SUCCESS. The procedure completed successfully.

KER$_BAD_STATE. A port specified to SET_USER contains unreceived messages or has an incomplete ACCEPT_CIRCUIT or CONNECT_CIRCUIT pending.

KER$_NO_ACCESS. An argument specified is not accessible to the calling program.

KER$_NO_SUCH_PORT. No port with the specified value can be found in the system or network, or the port is not owned by the current job, as required by SET_USER.

Authorization Service Utility Procedures

The procedures described in this section maintain the authorization database. Table 11-3 summarizes these procedures.

Table 11-3. Authorization Service Utility Procedures

Procedure	Purpose
AUTH_ADD_USER	adds a new user record to the authorization database.
AUTH_MODIFY_USER	modifies an existing user record in the authorization database.
AUTH_REMOVE_USER	removes an existing user record from the authorization database.
AUTH_SHOW_USER	returns authorization database information for the specified user or users.

Notes: To use these procedures, include the module $AUTHORIZE_UTILITY from the RTLOBJECT library in the compilation of your program.

These procedures all assume that the calling program has connected a circuit to the Authorization Service's AUTH$MAINTENANCE port.

The following Pascal types are declared in module $AUTHORIZE_UTILITY for use with the Authorization Service utility procedures:

```
TYPE
    auth$field_names = (auth$username_field,
        auth$nodename_field,auth$password_field,
        auth$uic_field, auth$userdata_field);
        { Authorization field names. }

    auth$fields = set of auth$field_names;
        { Authorization fields. }

    auth$username = varying_string(20);
        { Username string. }

    auth$password = varying_string(20);
        { Password string. }

    auth$nodename = varying_string(32);
        { Nodename string. }

    auth$userdata = varying_string(128);
        { User-specified data. }
```

AUTH_ADD_USER

The AUTH_ADD_USER procedure adds a new user record to the authorization database. This procedure requires that the caller be authorized with a system group UIC (that is, a UIC less than or equal to %X0008FFFF or [10,177777]). To use the procedure, you must include the module $AUTHORIZE_UTILITY in the compilation.

Call Format

```
ELN$AUTH_ADD_USER(
    status,
    circuit,
    username,
    nodename,
    password,
    uic,
    userdata
    )
```

Arguments

status. This optional argument is an INTEGER variable that receives the completion status of AUTH_ADD_USER.

circuit. This argument supplies a PORT value specifying the port connected in a circuit to the Authorization Service's AUTH$MAINTENANCE port.

username. This argument supplies a user name of type AUTH$USERNAME for the new user; it cannot be blank. The reserved name $ANY can be specified for the username argument, meaning that any user from the specified node that does not match one of the

explicit user names is authorized with the specified user identification code.

nodename. This argument supplies a node name of type AUTH$NODENAME for the node on which the new user is authorized; it can be blank. If nodename is specified, the database record represents a proxy authorization and the password is unused. If nodename is not specified, the database record represents a destination authorization. The reserved name $ANY can be specified for the nodename argument, meaning that any user with the specified name from any node that does not match one of the explicit node names is authorized with the specified user identification code.

password. This argument supplies a password of type AUTH$PASSWORD for the new user; it can be blank. If a destination authorization record is added, the password is stored with the record. Passwords are always stored in a scrambled form so that they cannot be read once they are stored.

uic. This argument is an INTEGER value that supplies the user identification code assigned to the new user.

userdata. This argument supplies an arbitrary string of user-specified data of type AUTH$USERDATA that is stored with the user record for use by applications.

AUTH_MODIFY_USER

The AUTH_MODIFY_USER procedure modifies an existing user record in the authorization database. This procedure requires that the caller be authorized with a system group UIC (that is, a UIC less than or equal to %X0008FFFF or [10,177777]). To use the procedure, you must include the module $AUTHORIZE_UTILITY in the compilation.

Call Format

```
ELN$AUTH_MODIFY_USER(
    status,
    circuit,
    username,
    nodename,
    new_fields,
    new_username,
    new_nodename,
    new_password,
    new_uic,
    new_userdata
    )
```

Arguments

status. This optional argument is an INTEGER variable that receives the completion status of AUTH_MODIFY_USER.

circuit. This argument supplies a PORT value specifying the port connected in a circuit to the Authorization Service's AUTH$MAINTENANCE port.

username. This argument supplies the user name of type AUTH$USERNAME for the record to be modified; it cannot be blank.

nodename. This argument supplies the name of the node of type AUTH$NODENAME on which the user is authorized.

new_fields. This argument, which can be blank, supplies a set that specifies which of the other fields are to be modified.

new_username. This argument supplies a new user name of type AUTH$USERNAME for the user; it cannot be blank. The reserved name $ANY can be

specified for the new_username argument, meaning that any user from the specified node that does not match one of the explicit user names is authorized with the specified user identification code. Note that if the user name is modified, the password must be reset as well.

- **new_nodename.** This argument supplies a new node name of type AUTH$NODENAME for the node on which the user is authorized; it can be blank. If new_nodename is specified, the database record represents a proxy authorization and the password is unused. If new_nodename is not specified, the database record represents a destination authorization. The reserved name $ANY can be specified for the new_nodename argument, meaning that any user with the specified name from any node that does not match one of the explicit node names is authorized with the specified user identification code.

  new_password. This argument supplies a new password of type AUTH$PASSWORD for the user; it can be blank. If a destination authorization record is added, the password is stored with the record. Passwords are always stored in a scrambled form so that they cannot be read once they are stored. Note that if the user name is modified, the password must be reset as well.

  new_uic. This argument is an INTEGER value that supplies the new user identification code assigned to the user.

  new_userdata. This argument supplies an arbitrary string of user-specified data of type AUTH$USERDATA that is stored with the user record for use by applications.

AUTH_REMOVE_USER

The AUTH_REMOVE_USER procedure removes an existing user record from the authorization database. This procedure requires that the caller be authorized with a system group UIC (that is, a UIC less than or equal to %X0008FFFF or [10,177777]). To use the procedure, you must include the module $AUTHORIZE_UTILITY in the compilation.

Call Format

```
ELN$AUTH_REMOVE_USER(
    status,
    circuit,
    username,
    nodename
    )
```

Arguments

status. This optional argument is an INTEGER variable that receives the completion status of AUTH_REMOVE_USER.

circuit. This argument supplies a PORT value specifying the port connected in a circuit to the Authorization Service's AUTH$MAINTENANCE port.

username. This argument supplies the user name of type AUTH$USERNAME of the user to be removed; it cannot be blank.

nodename. This argument supplies the name of the node of type AUTH$NODENAME on which the user is no longer authorized.

AUTH_SHOW_USER

The AUTH_SHOW_USER procedure returns authorization database information for the specified user or users. To use the procedure, you must include the module $AUTHORIZE_UTILITY in the compilation.

Note that the user-specified show_user procedure is only invoked if the specified user entry is found in the authorization database.

Call Format

```
ELN$AUTH_SHOW_USER(
    status,
    circuit,
    username,
    nodename,
    show_user
    )
```

Arguments

status. This optional argument is an INTEGER variable that receives the completion status of AUTH_SHOW_USER.

circuit. This argument supplies a PORT value specifying the port connected in a circuit to the Authorization Service's AUTH$MAINTENANCE port.

username. This argument supplies the user name of type AUTH$USERNAME of the user records to be accessed; it cannot be blank.

nodename. This argument supplies the name of the node of type AUTH$NODENAME from which the user is authorized. This argument should only be specified

as a non-null string if the proxy information for the specified user(s) is requested, in which case the proxy information is returned.

show_user. This argument supplies a procedure name of type AUTH$SHOW_USER_ROUTINE that identifies a user-specified routine to be invoked by AUTH_SHOW_USER.

Notes

This procedure calls the user-specified show_user procedure with the values of a specified user record or all the records in the authorization data file. To return all the records in the file, specify the input username parameter as the string '*'; the specified show_user procedure is then called once for each record in the file.

The procedure type declaration for AUTH$SHOW_USER_ROUTINE in module $AUTHORIZE_UTILITY is as follows:

```
PROCEDURE
    auth$show_user_routine(
        var username: auth$username;
        var nodename: auth$nodename;
        var uic: integer;
        var userdata: auth$userdata);
    PROCEDURE_TYPE;
```

Program Loader Utility Procedures

The procedures described in this section dynamically load and unload program images into a running VAXELN system after the initial system is built. Table 11-4 summarizes these procedures.

Table 11-4. Program Loader Utility Procedures

Procedure	Purpose
LOAD_PROGRAM	loads a specified image file into a currently running system.
UNLOAD_PROGRAM	unloads a specified program from a currently running system.

Note: To use these procedures, include the module $LOADER_UTILITY from the RTLOBJECT library in the compilation of your program.

LOAD_PROGRAM

The LOAD_PROGRAM procedure loads a specified image file into a currently running VAXELN system. After the image file is loaded, the CREATE_JOB procedure (described earlier in this chapter) is used to start the program running. To use the procedure, you must include the module $LOADER_UTILITY in the compilation.

Call Format

```
ELN$LOAD_PROGRAM(
    file_name,
    program_name,
    kernel_mode,
    start_with_debug,
    power_recovery,
    kernel_stack_size,
    initial_user_stack_size,
    message_limit,
    job_priority,
    process_priority,
    status
    )
```

Arguments

file_name. This argument supplies a string of up to 255 characters giving the name of the image file to be loaded into the system. The file is opened in the context of the caller, so the file name must be provided in sufficient detail to correctly identify the file. The file can reside on the system or on a remote node.

program_name. This argument supplies a string of up to 40 characters giving the name by which the program

will be known for the CREATE_JOB call. If the argument is specified as a null string, the image name supplied by the linker is used for the program name, and that name is returned in this argument.

kernel_mode. This argument is a BOOLEAN value specifying in which mode the program is to run. TRUE means kernel mode; FALSE (the default) means user mode.

start_with_debug. This argument is a BOOLEAN value specifying whether the debugger is to get control of the program when it is started. TRUE means the debugger is to get control; FALSE (the default) means the debugger is not to get control.

power_recovery. This argument is a BOOLEAN value specifying whether the job running the specified program is to be given the power recovery exception if the power fails on the system. TRUE means the job is to be given the power recovery exception; FALSE (the default) means the job is not to be given the power recovery exception.

kernel_stack_size. This argument is an INTEGER value that supplies the size, in pages, of the kernel mode stack for jobs running this program. User mode programs require at least 1 page (the default) of kernel stack.

initial_user_stack_size. This argument is an INTEGER value that supplies the initial size, in pages, of the user mode stack for jobs running this program. Programs require at least 1 page (the default) of user stack. This parameter is ignored for kernel mode programs.

message_limt. This argument is an INTEGER value that specifies the maximum number of messages the job port can contain; the default is 0.

job_priority. This argument is an integer from 0 to 31 that specifies the starting job priority for this program; the default is 16.

process_priority. This argument is an integer from 0 to 15 that specifies the starting process priority for this program; the default is 8.

status. This optional argument is an INTEGER variable that receives the completion status.

UNLOAD_PROGRAM

The UNLOAD_PROGRAM procedure unloads a specified program from a currently running VAXELN system. To use the procedure, you must include the module $LOADER_UTILITY in the compilation.

Call Format

```
ELN$UNLOAD_PROGRAM(
    program_name,
    status
    )
```

Arguments

program_name. This argument supplies a string of up to 40 characters identifying the program to be unloaded.

status. This optional argument is an INTEGER variable that receives the completion status.

Exit Utility Procedures

The procedures described in this section establish and delete an exit handler to perform cleanup operations following the termination of a job with the EXIT procedure. Table 11-5 summarizes these procedures.

Table 11-5. Exit Utility Procedures

Procedure	Purpose
CANCEL_EXIT_HANDLER	deletes a specific exit handler routine.
DECLARE_EXIT_HANDLER	calls an exit handler routine defined by the program.

Note: To use these procedures, include the module $EXIT_UTILITY from the RTLOBJECT library in the compilation of your program.

CANCEL_EXIT_HANDLER

The CANCEL_EXIT_HANDLER procedure allows you to cancel an exit handler (identified by the exit handler and an associated context value), which was enabled by DECLARE_EXIT_HANDLER. To use the procedure, you must include the module $EXIT_UTILITY in the compilation.

Call Format

```
ELN$CANCEL_EXIT_HANDLER(
    exit_handler,
    exit_context
    )
```

Arguments

exit_handler. This argument supplies a procedure name of type ELN$EXIT_HANDLER that identifies the exit handler routine to be cancelled.

exit_context. This optional argument is a variable of type ↑ANYTYPE. This variable must exactly match the exit_context variable used in the DECLARE_EXIT_HANDLER call in order for the proper handler to be cancelled.

DECLARE_EXIT_HANDLER

The DECLARE_EXIT_HANDLER procedure allows you to declare an exit handler for a program. The named exit handler procedure is called upon the termination of a job with the EXIT procedure. To use the procedure, you must include the module $EXIT_UTILITY in the compilation.

Call Format

```
ELN$DECLARE_EXIT_HANDLER(
    exit_handler,
    exit_context
    )
```

Arguments

exit_handler. This argument supplies a procedure name of type ELN$EXIT_HANDLER that identifies an exit handler routine to be called upon the termination of a job with the EXIT procedure.

exit_context. This optional argument is a variable of type ↑ ANYTYPE that will be passed to the specified exit handler routine when it is invoked.

Notes

The procedure type declaration for ELN$EXIT_HANDLER in module $EXIT_UTILITY is as follows:

```
PROCEDURE
    eln$exit_handler(
        var exit_context: [optional] ↑ anytype);
    PROCEDURE_TYPE;
```

MUTEX Data Type

The MUTEX data type is provided as an optimization of binary semaphores. To use the type and its associated procedures, you must include the module $MUTEX from the library RTLOBJECT in the compilation.

Mutex Operations

The general meanings of mutex operations are identical to the comparable operations on binary semaphores, with one important difference: When a process locks a mutex, to gain access to a shared resource, it does not have to call a WAIT procedure unless some other process has already locked the mutex. This results in a very significant improvement in efficiency compared with simply calling the WAIT and SIGNAL procedures with binary semaphores.

The operations on mutexes are as follows:

- The CREATE_MUTEX procedure initializes a mutex (initially unlocked) and creates its associated semaphore.
- The INITIALIZE_AREA_MUTEX initializes a mutex that uses an AREA object as the synchronization object.
- The LOCK_MUTEX procedure locks a mutex (used in lieu of WAIT_ANY or WAIT_ALL).
- The UNLOCK_MUTEX procedure unlocks a mutex (used in lieu of SIGNAL).
- The DELETE_MUTEX procedure deletes the semaphore created for a mutex.

Each of these procedures takes a MUTEX variable as one of its arguments (or as its only argument).

As an example of the use of mutex operations, consider a program in which several concurrent processes will write values in a device register:

```
MODULE mutexample;
INCLUDE $MUTEX;
.
.
VAR playmutex: MUTEX;
.
.
PROGRAM main;
   BEGIN
     CREATE_MUTEX(playmutex);
     .
     . { CREATE_PROCESS calls to activate
       play subprocesses. }
     .
   END; { End of main program. }
PROCESS_BLOCK play;.
   BEGIN
     LOCK_MUTEX(playmutex);
     { Prevent other subprocesses from writing the
     device registers until this one is done. }
     .
     . { WRITE_REGISTER calls to load device
       registers. }
     .
     UNLOCK_MUTEX(playmutex);
     { Allow other subprocesses access to registers. }
   END; { End of process block. }
END; { End of module mutexample. }
```

Internal Representation of Mutexes

A MUTEX value is represented internally as a 6-byte record containing a 16-bit counter and a SEMAPHORE value. The counter is initialized to −1 and the SEMAPHORE value to a binary semaphore by CREATE_MUTEX. The counter is then incremented and decremented (using the ADD_INTERLOCKED routine) by LOCK_MUTEX and UNLOCK_MUTEX, respectively.

If LOCK_MUTEX increments the counter and the result is greater than 0, another process has already locked the mutex, so LOCK_MUTEX calls a WAIT procedure to wait for the semaphore. If UNLOCK_MUTEX decrements the counter and the result is greater than or equal to 0, another process is waiting for the mutex, so UNLOCK_MUTEX calls SIGNAL to signal the semaphore.

When a mutex is deleted with the DELETE_MUTEX procedure, the counter is set to 0, indicating that the mutex is locked. A subsequent call to LOCK_MUTEX will then call a WAIT procedure and fail with the status KER$_BAD_VALUE.

Mutex Procedures

The procedures described in this section perform operations on mutexes. Table 11-6 summarizes these procedures.

Table 11-6. Mutex Procedures

Procedure	Purpose
CREATE_MUTEX	initializes a mutex and creates its associated semaphore.
DELETE_MUTEX	deletes the semaphore created for a mutex.
INITIALIZE_AREA_MUTEX	initializes a new mutual exclusion semaphore that uses an AREA object as the synchronization object.
LOCK_MUTEX	locks a mutex.
UNLOCK_MUTEX	unlocks a mutex.

Note: To use these procedures, include the module $MUTEX from the RTLOBJECT library in the compilation of your program.

CREATE_MUTEX

The CREATE_MUTEX procedure initializes a MUTEX variable for use in guarding the access to a shared variable or other shared resource. The initial state is "unlocked." The procedure creates a SEMAPHORE object and stores its identifying value in one of the MUTEX variable's fields. To use the procedure, you must include the module $MUTEX in the compilation.

Call Format

```
ELN$CREATE_MUTEX(
    mutex,
    status
    )
```

Arguments

mutex. This argument is a MUTEX variable that receives the new MUTEX value.

status. This optional argument is an INTEGER variable that receives the completion status.

DELETE_MUTEX

The DELETE_MUTEX procedure deletes the semaphore associated with a MUTEX variable. To use the procedure, you must include the module $MUTEX in the compilation.

Call Format

```
ELN$DELETE_MUTEX(
    mutex,
    status
    );
```

Arguments

mutex. This argument is a variable of type MUTEX. It is not modified by the procedure. The semaphore identified by the variable's semaphore field is deleted from the system.

status. This optional argument is an INTEGER variable that receives the completion status.

INITIALIZE_AREA_MUTEX

The INITIALIZE_AREA_MUTEX procedure initializes a new mutual exclusion semaphore that uses an AREA object as the synchronization object. To use the procedure, you must include the module $MUTEX in the compilation.

Call Format

```
ELN$INITIALIZE_AREA_MUTEX(
    mutex,
    area,
    status
    )
```

Arguments

mutex. This argument is a MUTEX variable that receives the new MUTEX value.

area. This argument is an AREA variable that receives the identifier of the new area.

status. This optional argument is an INTEGER variable that receives the completion status.

LOCK_MUTEX

The LOCK_MUTEX procedure locks a mutex. To use the procedure, you must include the module $MUTEX in the compilation.

Call Format

```
ELN$LOCK_MUTEX(mutex)
```

Arguments

mutex. The argument is a variable of type MUTEX.

UNLOCK_MUTEX

The UNLOCK_MUTEX procedure unlocks a mutex. To use the procedure, you must include the module $MUTEX in the compilation.

Call Format

```
ELN$UNLOCK_MUTEX(mutex)
```

Arguments

mutex. The argument is a variable of type MUTEX.

Chapter 12
Interjob Communication

This chapter discusses messages and ports as they relate to interjob communication, including the kernel services relating to message transmission. In addition, this chapter discusses interjob data sharing using AREA objects and the related kernel services. Finally, the memory allocation procedures and stack utility procedures provided by VAXELN are discussed.

Messages and Ports

The VAXELN kernel provides a MESSAGE object to describe a block of memory that can be moved from one job's virtual address space to another's. The block of memory is called *message data* and is allocated dynamically by the kernel from physically contiguous, page-aligned blocks of memory. A MESSAGE object and its associated message data are both created by calling the CREATE_MESSAGE kernel service.

Message data is mapped into a job's P0 virtual address space, so it is potentially accessible to all the processes in the job. If a message is sent to a job on the sending job's local node, the kernel unmaps the message data from the sending job's virtual address space and remaps it into the receiver's space.

If a message is sent to a remote node, the kernel again unmaps the message data, but it remaps it into the appropriate network device driver job to send the message to the remote system. The reverse operations

then cause the message data to be remapped in the receiver's space.

A PORT object represents a system-maintained message queue that is created with the CREATE_PORT kernel service. PORT object values identify unique destinations for messages; they can be passed as arguments, sent in messages, or obtained from the RECEIVE procedure.

To facilitate communication between jobs, message ports can be given names. These names are created with the CREATE_NAME kernel service. Names have their own data type in order to establish ownership of the name; that way, only the process that creates a name, or some other process in the same job, can delete it. Names can be either local to a node or universal. A local name is guaranteed to be unique within the local node. Universal names are guaranteed to be unique throughout the entire local area network.

Sending Messages

To send a message, you declare a pointer to the type of data you want to send, supply the pointer to CREATE_MESSAGE, use the pointer to fill in the message data (the size of the message data is implied by the pointer), and supply the MESSAGE and PORT object values to the SEND procedure. For example:

```
VAR dptr: ↑INTEGER;
    msg: MESSAGE;
    dest: PORT;
BEGIN
    CREATE_MESSAGE(msg,dptr);
    dptr↑ := 512;
    SEND(msg,dest);
END.
```

The SEND procedure removes the message data from your job and places the MESSAGE object value in the destination port.

Receiving Messages

The receiver process waits for a message to arrive on its port and then uses the RECEIVE procedure to obtain it. The RECEIVE procedure automatically maps the message data into the receiver's address space, returns a MESSAGE object value for the receiver's use, and optionally returns the values of the reply port and destination port. To reply, the receiver formulates an answer and sends a reply to the reply port.

Note that any expedited data messages queued to a port are received by the RECEIVE procedure before any normal data messages are received.

Datagrams and Circuits

Two methods are used to transmit messages:

- The *datagram* method is used when messages are sent between unconnected message ports.
- The *circuit* method is used when messages are sent between two ports connected via a circuit.

The datagram method cannot guarantee that a message is actually received at the destination; however, it does guarantee that received messages are correct. In addition, two messages sent to the same destination port can possibly arrive in a different order.

In contrast, messages sent through circuits are guaranteed to be delivered (if the physical connection is intact) and to be delivered in the same sequence in which they are sent.

Using circuits, messages can have any length, and, if the transmission is across the network, the network services will divide the message into segments of the proper length, transmit the segments in sequence, and reassemble them at the destination node.

In addition, the OPEN procedure permits you to "open" a circuit as if it were a file and to use the Pascal I/O routines (such as READ and WRITE) to transmit messages. (See Chapter 15, "Input and Output," for descriptions of the OPEN procedure and all Pascal I/O routines.)

Programming with Circuits

Circuits are established between two ports by the CONNECT_CIRCUIT and ACCEPT_CIRCUIT procedures. Options of these procedures allow you to control the flow of messages through a circuit; that is, you can prevent a sending process from sending too many messages to a slower receiving process.

A process that wants to establish a circuit calls CONNECT_CIRCUIT and designates a destination port in another process. A special connection request message is automatically sent to the designated port. For example:

```
CONNECT_CIRCUIT(myport,
   DESTINATION_NAME := 'request_server');
```

Here, myport is a port in the calling process that will form its half of the circuit. The destination name is specified by the string 'request_server', which is translated automatically by CONNECT_CIRCUIT to designate the destination port.

Elsewhere, an ACCEPT_CIRCUIT call causes a process to wait for a connection request message on the designated port.

For example:

```
VAR
   server : NAME;
   receiver_port, connect_port: PORT;

CREATE_PORT(receiver_port, LIMIT : = 10);
CREATE_PORT(connect_port);
CREATE_NAME(server,'request_server',
   receiver_port);

ACCEPT_CIRCUIT(receiver_port,
   CONNECT : = connect_port);

{ Wait for a connection request. When the wait is
satisfied, a circuit is established between the
requestor and connect_port. }
```

At this point, the acceptor can take a variety of actions to communicate with the requestor, such as creating a subprocess to continue the dialog and passing it the port value (connect_port) representing its half of the circuit. The ACCEPT_CIRCUIT procedure can notify you of error conditions, such as an unreceived message in receive_port or another connection request for which acceptance is still pending.

Circuits are broken when either partner calls the DISCONNECT_CIRCUIT procedure. The SEND and RECEIVE procedures both notify their callers if the designated port was disconnected.

For more information on messages, ports, and circuits, see the *VAXELN User's Guide*.

The kernel services relating to message transmission are described in detail in the following section. A brief description of each procedure is given, followed by the VAXELN Pascal call format, arguments, and status values.

Kernel Services for Message Transmission

The kernel services described in this section relate to transmitting messages between processes, jobs, and ports. Table 12-1 summarizes these procedures.

Table 12-1. Kernel Services for Message Transmission

Procedure	Purpose
ACCEPT_CIRCUIT	establishes a circuit between two ports.
CONNECT_CIRCUIT	connects a port to a specified destination port.
CREATE_MESSAGE	creates a message and its associated message data.
CREATE_NAME	creates a name for a port.
CREATE_PORT	creates a message port.
DISCONNECT_CIRCUIT	breaks the circuit connection between two ports.
JOB_PORT	returns the current job port.
RECEIVE	receives a message from a port.
SEND	sends a message to a port.
TRANSLATE_NAME	returns a value identifying a named port.

ACCEPT_CIRCUIT

The ACCEPT_CIRCUIT procedure causes the invoking process to wait for a circuit connection. When the wait is satisfied (that is, on successful completion), the circuit is established between two ports.

Call Format

```
ACCEPT_CIRCUIT (
   source_port,
   CONNECT : = connect_port,
   FULL_ERROR : = flag,
   ACCEPT_DATA : = accdata,
   CONNECT_DATA : = conndata,
   STATUS : = stat
   )
```

Arguments

source_port. This argument supplies the value of the port on which to wait for a connection request. Unless connect_port is present, this port also forms the invoker's half of the circuit. If, during the call, this port receives a message that is not a connection request, the message is ignored.

connect_port. This optional argument supplies a different port, which is used for the actual connection; if it is absent, the source_port value is used for the connection. This argument need only be specified if additional connections need to be accepted before previous ones are disconnected.

flag. This optional argument supplies a BOOLEAN value to enable or disable the implicit wait caused when the partner port is full. The default is FALSE, meaning that the sender waits if the partner is full. If

TRUE is supplied, an error status or the corresponding exception occurs with SEND when you attempt to send a message and the partner's port is full.

accdata. This optional argument supplies a VARYING_STRING(16) value that is passed to the process requesting the circuit connection (that is, the requesting process receives this value in the ACCEPT_DATA parameter of its CONNECT_CIRCUIT call).

conndata. This optional argument is a variable of type VARYING_STRING(16) that receives data passed by the requesting process in the CONNECT_DATA parameter of its CONNECT_CIRCUIT call.

stat. This optional argument is an INTEGER variable that receives the completion status of ACCEPT_CIRCUIT.

Status Values

KER$_SUCCESS. The procedure completed successfully.

KER$_BAD_STATE. A port specified to ACCEPT_CIRCUIT contains unreceived messages or has an incomplete CONNECT_CIRCUIT or ACCEPT_CIRCUIT pending.

KER$_CONNECT_PENDING. A CONNECT_CIRCUIT is pending, and the port cannot be used for another purpose until the connection has completed.

KER$_NO_ACCESS. An argument specified is not accessible to the calling program.

KER$_NO_SUCH_PORT. No port with the specified value can be found in the system or network, or the port is not owned by the current job, as required by ACCEPT_CIRCUIT.

CONNECT_CIRCUIT

The CONNECT_CIRCUIT procedure connects a port to a specified destination port. If the process receiving the connection request accepts it, the two ports are bound together in a circuit. The destination port can be specified either by name or by PORT value.

Call Format

```
CONNECT_CIRCUIT(
    port,
    DESTINATION_PORT : = dest_port,
    DESTINATION_NAME : = string,
    FULL_ERROR : = flag,
    CONNECT_DATA : = conndata,
    ACCEPT_DATA : = accdata,
    STATUS : = stat
    )
```

Arguments

port. This argument supplies a PORT value that will form the caller's half of the circuit.

dest_port. This optional argument supplies a PORT value giving the destination for the connection request message. (Such a PORT value can be obtained from the reply_port argument of RECEIVE).The argument can be omitted only if a destination name is supplied by the following argument.

string. This optional argument supplies the destination for the connection request message as a character-string name, usually a name established by the CREATE_NAME procedure. If the destination is specified this way, via a NAME object, string is automatically translated to a destination port. If

dest_port is also specified, it overrides this argument. (Either this argument or dest_port must be present.) The string can also have the forms

```
nodename::local_port_name
```

```
nodenumber::local_port_name
```

for connection to a port in a VAXELN system (where the local_port_name is a local name established on the VAXELN node), or the form

```
nodenumber::object
```

for connection to a DECnet–VAX (VAX/VMS) system (where object is the name of the object on the non-VAXELN DECnet system that will handle the connection).

For more information about connections to non-VAXELN systems, see the *VAXELN User's Guide.*

flag. This optional argument supplies a BOOLEAN value to enable or disable the implicit wait performed (with SEND) when the partner port is full. The default is FALSE, meaning that the sender waits until the partner port is not full; if TRUE is specified, SEND returns an error status or raises the corresponding exception if the partner port is full.

conndata. This optional argument supplies data to the process receiving the connection request. The data type is VARYING_STRING(16).

accdata. This optional argument is a variable of type VARYING_STRING(16) that receives any data supplied by the accepting process in its ACCEPT_CIRCUIT call.

stat. This optional argument is an INTEGER variable that receives the completion status of CONNECT_CIRCUIT.

Status Values

KER$_SUCCESS. The procedure completed successfully.

KER$_BAD_STATE. A port specified to CONNECT_CIRCUIT contains unreceived messages or has an incomplete CONNECT_CIRCUIT or ACCEPT_CIRCUIT pending.

KER$_CONNECT_TIMEOUT. The connection request was not accepted by the destination within the connection timeout limit. The connection timeout time can be set by the System Builder utility.

KER$_NO_ACCESS. An argument specified is not accessible to the calling program.

KER$_NO_DESTINATION. Neither a destination port value nor port name was specified in the procedure call.

KER$_NO_SUCH_NAME. The procedure call specified a NAME value for which there is no translation.

KER$_NO_SUCH_PORT. No port with the specified value can be found in the system or network, or the port is not owned by the current job, as required by CONNECT_CIRCUIT.

CREATE_MESSAGE

The CREATE_MESSAGE procedure creates a MESSAGE object and allocates storage for its associated message data.

Call Format

```
CREATE_MESSAGE(
   message,
   data_pointer,
   STATUS : = stat
   )
```

Arguments

message. This argument is a MESSAGE variable that receives the identifier of the new MESSAGE object.

data_pointer. This argument supplies a pointer variable that will identify the message's data. It can have any type except ↑ANYTYPE. The procedure allocates storage of the base type's size and sets the pointer to identify it. The returned pointer value is valid in the current job; it becomes invalid if the message is sent or deleted.

stat. This optional argument is an INTEGER variable that receives the completion status of CREATE_MESSAGE.

Status Values

KER$_SUCCESS. The procedure completed successfully.

KER$_NO_MEMORY. There was not enough physical memory to create the message data area.

KER$_NO_OBJECT. No free job object table entries are currently available. There are a maximum of 1024 object table entries per job.

KER$_NO_POOL. No free system pool is currently available. The size of the system pool can be set by the System Builder utility.

CREATE_NAME

The CREATE_NAME procedure creates a NAME object that refers to a specified port. Names created by this procedure are guaranteed to be unique within the specified name space (local or universal). If you attempt to create a name that is not unique, the NAME object is not created, and an error status is returned.

Call Format

```
CREATE_NAME(
    name,
    name_string,
    port_value,
    TABLE : = table,
    STATUS: = stat
    )
```

Arguments

name. This argument is a NAME variable that receives the identifier of the new NAME object.

name_string. This argument supplies the name (as a 1–31-character string).

port_value. This argument supplies the value of the port being named.

table. This optional argument supplies the enumerated value NAME$LOCAL, NAME$UNIVERSAL, or NAME$BOTH. It specifies that the new name is either local (valid only in this system, or node), universal (valid throughout the application, or on any node), or both. NAME$LOCAL is the default. If the system does not contain the Network Service, all names are placed in the local name table.

stat. This optional argument is an INTEGER variable that receives the completion status of CREATE_NAME.

Status Values

KER$_SUCCESS. The procedure completed successfully.

KER$_BAD_LENGTH. A string argument was too long.

KER$_DUPLICATE. The CREATE_NAME procedure was called with a name string that is a duplicate of an existing name.

KER$_NO_ACCESS. An argument specified is not accessible to the calling program.

KER$_NO_OBJECT. No free job object table entries are currently available. There are a maximum of 1024 object table entries per job.

KER$_NO_POOL. No free system pool is currently available. The size of the system pool can be set by the System Builder utility.

CREATE_PORT

The CREATE_PORT procedure creates a message port and optionally specifies its maximum message capacity.

Call Format

```
CREATE_PORT(
   port,
   LIMIT : = int,
   STATUS : = stat
   )
```

Arguments

port. This argument is a PORT variable that receives the identifier of the new PORT object.

int. This optional argument is an integer expression that supplies the maximum number of messages that can be queued to the port at one time. If the maximum is exceeded and the port is connected in a circuit, the sending process waits until a message is received from

the port. If the port is not connected in a circuit, further messages are lost. The default value is 4.

stat. This optional argument is an INTEGER variable that receives the completion status of CREATE_PORT.

Status Values

KER$_SUCCESS. The procedure completed successfully.

KER$_NO_ACCESS. An argument specified is not accessible to the calling program.

KER$_NO_OBJECT. No free job object table entries are currently available. There are a maximum of 1024 object table entries per job.

KER$_NO_POOL. No free system pool is currently available. The size of the system pool can be set by the System Builder utility.

KER$_NO_PORT. No free system port table entries are currently available. The size of the system port table can be set by the System Builder utility.

DISCONNECT_CIRCUIT

The DISCONNECT_CIRCUIT procedure is used to break the circuit connection between two ports. If any process is waiting for either port in the circuit, its wait condition is satisfied. A request for connection can be rejected by first calling ACCEPT_CIRCUIT and then calling DISCONNECT_CIRCUIT.

Call Format

```
DISCONNECT_CIRCUIT(
    port_value,
    STATUS: = stat
    )
```

Arguments

port_value. This argument supplies a PORT value representing the caller's half of the circuit.

stat. This optional argument is an INTEGER variable that receives the completion status of DISCONNECT_CIRCUIT.

Status Values

KER$_SUCCESS. The procedure completed successfully.

KER$_BAD_STATE. A port specified to DISCONNECT_CIRCUIT was not connected.

KER$_NO_ACCESS. An argument specified is not accessible to the calling program.

KER$_NO_SUCH_PORT. No port with the specified value can be found in the system or network, or else the port is not owned by the current job, as required by DISCONNECT_CIRCUIT.

JOB_PORT

The procedure JOB_PORT returns a PORT value identifying the caller's job port. A unique job port is created whenever a job is created.

Call Format

```
JOB_PORT(
    port,
    STATUS : = stat
    )
```

Arguments

port. This argument is a PORT variable that receives a PORT value identifying the caller's job port.

stat. This optional argument is an INTEGER variable that receives the completion status of JOB_PORT.

Status Values

KER$_SUCCESS. The procedure completed successfully.

KER$_NO_ACCESS. An argument specified is not accessible to the calling program.

RECEIVE

The RECEIVE procedure removes a MESSAGE object from the designated message port and maps the message data into the receiver job's virtual address space.

Call Format

```
RECEIVE(
    message,
    data_ptr,
    port,
    SIZE : = size,
    DESTINATION : = dest_port,
    REPLY : = reply_port,
    STATUS : = stat
    )
```

Arguments

message. This argument is a MESSAGE variable that receives the MESSAGE value identifying the next message, if there is one in the port.

data_ptr. This argument is a pointer variable that receives a pointer value identifying the message data. The pointer value is valid only in the current job and becomes invalid if the message is sent or deleted. The variable you supply can be a pointer to any type. (Presumably, its type is the same one used by the sender to create the message.)

port. This argument supplies the PORT value of the port from which to retrieve the message.

size. This optional argument is an INTEGER value that receives the size in bytes of the message data.

dest_port. This optional argument is a PORT variable that receives the value of the destination port. Normally, this is the same value supplied by the sender for the receiver's port. It is available, and returns a different value, only for the internal interface between the kernel and the Network Service.

reply_port. This optional argument is a PORT variable that receives the value of the reply port. Note that this value is not set properly by RECEIVE if the port is connected in a circuit.

stat. This optional argument is an INTEGER variable that receives the completion status of RECEIVE.

Status Values

KER$_SUCCESS. The procedure completed successfully.

KER$_EXPEDITED. The procedure completed successfully, and the received message is an expedited message.

KER$_CONNECT_PENDING. A CONNECT_CIRCUIT is pending, and the port cannot be used for another purpose until the connection has completed.

KER$_DISCONNECT. The circuit was disconnected by the partner process.

KER$_NO_ACCESS. An argument specified is not accessible to the calling program.

KER$_NO_MESSAGE. No unreceived messages are currently in the port.

KER$_NO_OBJECT. No free job object table entries are currently available. There are a maximum of 1024 object table entries per job.

KER$_NO_SUCH_PORT. No port with the specified value can be found in the system or network, or the port is not owned by the current job, as required by RECEIVE.

KER$_NO_VIRTUAL. No free virtual address space (to map the message data) is currently available for the job. The size of the job's virtual address space can be set using the System Builder.

SEND

The SEND procedure removes a message's message data from the sender's address space and then places the MESSAGE object that describes the data in the destination's message port.

Call Format

```
SEND(
   message,
   destination,
   SIZE : = size,
   REPLY : = reply_port,
   EXPEDITE : = expedite,
   STATUS: = stat
   )
```

Arguments

message. The first argument supplies the MESSAGE value identifying the message to send. After the operation, any pointers to the message data are no longer valid.

destination. This argument supplies the PORT value identifying the destination port; if the message is being sent through a circuit, this port is the sender's half, and the message arrives at the receiver's half.

size. This optional argument is an INTEGER value that supplies the length in bytes of the message data to be sent; if it is omitted, the size of the originally created message data is the default. If size is specified, its value must be equal to or less than the original message data size.

reply_port. This optional argument is a PORT value identifying the reply port. If it is not specified, the kernel supplies the value of the sender's job port.

expedite. This optional argument supplies a BOOLEAN value stating whether to expedite the message. The default is FALSE. An expedited message bypasses the normal flow-control mechanism and can be sent even if the receiving port already has its maximum number of messages. The message is received by the port before any normal data messages. The size of an expedited message must not exceed 16 bytes.

stat. This optional argument is an INTEGER variable that receives the completion status of SEND.

Status Values

KER$_SUCCESS. The procedure completed successfully.

KER$_BAD_MESSAGE_SIZE. The message data is too large to be sent to the destination port. This can occur for the following reasons:

- The message is being sent to a remote port not connected in a circuit. The maximum message data size that can be sent as a datagram to a remote port is the System Builder's *Network Segment Size* minus 32. The default segment size is 576, so the maximum size remote datagram in the default case is 544 bytes.
- The message is being expedited. The maximum message data size that can be sent as an expedited message is 16 bytes.

KER$_BAD_TYPE. The first argument is not of type MESSAGE.

KER$_BAD_VALUE. The message or size argument is invalid or the message argument refers to a deleted mesage.

KER$_CONNECT_PENDING. A CONNECT_CIRCUIT is pending, and the port cannot be used for another purpose until the connection has completed.

KER$_COUNT_OVERFLOW. The destination port is full (with circuits, raised if the FULL_ERROR parameter in the ACCEPT_CIRCUIT or CONNECT_CIRCUIT procedure was TRUE).

KER$_DISCONNECT. The circuit was disconnected by the partner process.

KER$_NO_ACCESS. An argument specified is not accessible to the calling program.

KER$_NO_SUCH_PORT. No port with the specified value can be found in the system or network, or the port is not owned by the current job as required by SEND with circuits.

TRANSLATE_NAME

The TRANSLATE_NAME procedure returns a value identifying a named port. The specified name string is used to search for a NAME object with a matching string. If the NAME object is found, a value for the name's associated port is returned.

Call Format

```
TRANSLATE_NAME(
   port,
   name_string,
   selector,
   STATUS : = stat
   )
```

Arguments

port. This argument is a PORT variable that receives the value of the associated message port.

name_string. This argument is a character string that supplies the name of the port. Name strings are not case sensitive; uppercase and lowercase versions of the same name mean the same thing.

selector. This argument specifies which name table (local or universal) is to be searched. Possible values are values of the predeclared enumerated type NAME_TABLE:

- NAME$LOCAL specifies that only the local name table is searched.
- NAME$UNIVERSAL specifies that only the universal name table is searched.

- NAME$BOTH specifies that the local name table is searched first, followed by the universal table. The search ends as soon as a match is found.

stat. This optional argument is an INTEGER variable that receives the completion status of TRANSLATE_NAME.

Status Values

KER$_SUCCESS. The procedure completed successfully.

KER$_NO_ACCESS. An argument specified is not accessible to the calling program.

KER$_NO_SUCH_NAME. The translation for the specified name string cannot be found.

Interjob Data Sharing

Although messages are the recommended means of communicating between jobs, VAXELN provides AREA objects as a means of sharing data among jobs on a single VAXELN node. An AREA object represents a shareable region of memory and an associated binary semaphore, which can be used by the sharing jobs (and subprocesses) to synchronize access to the area's data. The synchronization must be done with kernel procedures or with the mutex routines, as discussed in Chapter 11, "Subprocesses and Synchronization."

The CREATE_AREA kernel service is used for two purposes:

- To create a new AREA object with a specified name
- To access an existing named area

In both cases, the procedure maps the area's data region into the calling job's P0 address space. If the original (creating) caller specifies a specific P0 virtual address for the area, it will be mapped into all jobs at this address. This feature makes the area *not* position independent; the sharing jobs can place real, fixed-pointer values in the region and they mean the same thing in each sharer's address space.

If a virtual address is not specified in the original call, the CREATE_AREA procedure will allocate a free P0 base address. Since the area could be in a different place in each sharer's space, fixed-pointer values cannot be used in the area. This is the typical case, with the area being used to hold one data structure.

The data region for an AREA object is allocated from physically contiguous 512-byte pages of memory and is mapped into the creating job's P0 virtual address space.

The region always occupies an integral number of memory pages and is aligned on a page boundary.

Note that areas with a size of zero are valid and represent only the semaphore.

The kernel services relating to interjob data sharing are described in detail in the following section. A brief description of each procedure is given, followed by the VAXELN Pascal call format, arguments, and status values.

Kernel Services for Interjob Data Sharing

The kernel services described in this section relate to sharing data among jobs on a single VAXELN node. Table 12-2 summarizes these procedures.

Table 12-2. Kernel Services for Interjob Data Sharing

Procedure	Purpose
CREATE_AREA	creates a new area or maps an existing area of memory into the creating job's P0 virtual address space.

CREATE_AREA

The CREATE_AREA procedure creates a new area or maps an existing area of memory into the creating job's P0 virtual address space.

Call Format

```
CREATE_AREA(
    area,
    data_pointer,
    area_name,
    VIRTUAL: = base_va,
    STATUS : = stat
    )
```

Arguments

area. This argument is an AREA variable that receives the identifier of the new AREA object.

data_pointer. This argument supplies a pointer variable for the base address of the area. It can have any type except ↑ ANYTYPE. The procedure creates a data area of the base type's size and validates the pointer to identify it.

area_name. This argument supplies the name for the area (as a 1–31-character string).

base_va. This optional argument supplies the base virtual address where the area is to be placed; it must be in P0 space.

stat. This optional argument is an INTEGER variable that receives the completion status of CREATE_AREA.

Status Values

KER$_SUCCESS. The procedure completed successfully.

KER$_BAD_VALUE. The area_name argument has a bad length, the base virtual address or ending address is not in P0 space, or the virtual address is specified and does not match the area's specified virtual address.

KER$_NO_ACCESS. An argument specified is not accessible to the calling program.

KER$_NO_MEMORY. There were not enough memory pages to complete the operation.

KER$_NO_OBJECT. No free job object table entries are currently available. There are a maximum of 1024 object table entries per job.

KER$_NO_POOL. No free system pool is currently available. The size of the system pool can be set by the System Builder utility.

KER$_NO_VIRTUAL. The necessary virtual address range is not available in the calling job's virtual address space.

Memory Allocation Procedures

The procedures described in this section relate to allocating and freeing memory. Table 12-3 summarizes these procedures.

Table 12-3. Memory Allocation Procedures

Procedure	Purpose
ALLOCATE_MEMORY	allocates physical VAX memory to the calling job.
FREE_MEMORY	frees a region of physical VAX memory previously allocated.
MEMORY_SIZE	scans the kernel memory data base to determine the free and largest block sizes.

Note: To use MEMORY_SIZE, include the module $KERNEL from the RTLOBJECT library in the compilation of your program.

ALLOCATE_MEMORY

The ALLOCATE_MEMORY kernel procedure allocates physical memory and maps it into the virtual address space of the job that calls it. The memory allocation can be specified to start at a given virtual address or at a given physical address, or both.

Call Format

```
ALLOCATE_MEMORY(
   mem_pointer,
   size,
   VIRTUAL : = virtual_address,
    PHYSICAL : = physical_address,
   STATUS : = stat
   )
```

Arguments

mem_pointer. This argument is a pointer variable that receives a pointer (↑ ANYTYPE) to the first location of the allocated memory. The received value is the virtual address.

size. This argument supplies an INTEGER value giving the number of bytes of memory to allocate. The value supplied is increased to the next multiple of 512.

virtual_address. This optional argument is a pointer value that supplies the starting virtual address of the allocated memory. The value is truncated if necessary to address a 512-byte page boundary. If this argument is omitted, the memory is allocated using any available contiguous address space in the calling job's P0 region. If the argument is present, allocation is attempted at the specified location in P0 or P1.

physical_address. This optional argument is an INTEGER value that supplies the starting physical address of the allocated memory. It is truncated if necessary to address a 512-byte page boundary. If it is omitted, the allocated memory comes from the system's pool of free memory.

stat. This optional argument is an INTEGER variable that receives the completion status of ALLOCATE_MEMORY.

Notes

For most purposes, you should allocate memory with the NEW procedure, which then calls the ALLOCATE_MEMORY procedure.

Caution should be exercised when the PHYSICAL parameter is used. The kernel maintains a list of all the pages of physical memory in the system that are free. If the ALLOCATE_MEMORY procedure is called without a PHYSICAL parameter specified, the kernel uses the list to determine what pages of memory can be allocated. If the PHYSICAL parameter is specified, however, the kernel does not consult the list. Instead, it assumes the calling program knows what pages are unused; for example, the memory associated with I/O registers or multi-ported memory. Since specifying the PHYSICAL parameter incorrectly could cause a program to overwrite currently allocated memory, the program must be running in kernel mode to use the PHYSICAL parameter.

Status Values

KER$_SUCCESS. The procedure completed successfully.

KER$_BAD_MODE. The physical_address argument was specified by a program that was not running in kernel

mode; kernel mode is required to allocate specific physical memory.

KER$_BAD_VALUE. The virtual_address argument is not in the job's address space.

KER$_NO_ACCESS. An argument specified is not accessible to the calling program.

KER$_NO_MEMORY. No free pages of physical memory are currently available.

KER$_NO_VIRTUAL. No free contiguous virtual address space is currently available for the process. The size of process virtual address space can be set using the System Builder utility.

FREE_MEMORY

The FREE_MEMORY kernel procedure frees a region of memory previously allocated by the ALLOCATE_MEMORY procedure. (Note that dynamically allocated memory is normally allocated with the NEW procedure and freed with DISPOSE.) Any pointers to the freed memory become invalid.

Call Format

```
FREE_MEMORY(
   size,
   virtual_address,
   STATUS: = stat
   )
```

Arguments

size. This is an integer expression that supplies the number of bytes of memory to be freed. This value is increased to the next 512-byte page.

virtual_address. This argument supplies the starting virtual address of the memory, as returned by ALLOCATE_MEMORY. This value is truncated to a 512-byte page address.

stat. This optional argument is an INTEGER variable that receives the completion status of FREE_MEMORY.

Status Values

KER$_SUCCESS. The procedure completed successfully.

KER$_BAD_VALUE. The virtual_address argument is not in the calling job's address space.

MEMORY_SIZE

The MEMORY_SIZE kernel procedure scans the kernel memory data base and returns, in 512-byte pages, the initial main memory, the current free memory, and the size of the largest, physically contiguous, block of free memory. While MEMORY_SIZE performs the memory scan, all other kernel operations are stopped. To use the procedure, you must include the module $KERNEL in the compilation.

Call Format

```
KER$MEMORY_SIZE(
   status,
   memory_size,
   free_size,
   largest_size
   )
```

Arguments

status. This optional argument is an INTEGER variable that receives the completion status of MEMORY_SIZE.

memory_size. This argument is an INTEGER variable that receives the size, in 512-byte pages, of the initial main memory.

free_size. This argument is an INTEGER variable that receives the size, in 512-byte pages, of the current free memory.

largest_size. This argument is an INTEGER variable that receives the size, in 512-byte pages, of the largest, physically contiguous, block of free memory.

Status Values

KER$_SUCCESS. The procedure completed successfully.

KER$_NO_ACCESS. An argument specified is not accessible to the calling program.

Stack Utility Procedures

The procedures described in this section explicitly manage the stack size during the execution of a program. Table 12-4 summarizes these procedures.

Table 12-4. Stack Utility Procedures

Procedure	Purpose
ALLOCATE_STACK	verifies the availability of a particular amount of stack space.
DEALLOCATE_STACK	trims the stack by up to the number of bytes specified.

Note: To use these procedures, include the module $STACK_UTILITY from the RTLOBJECT library in the compilation of your program.

ALLOCATE_STACK

The ALLOCATE_STACK procedure verifies that the process has the stack space you requested; if not, it allocates the stack space. To use the procedure, you must include the module $STACK_UTILITY in the compilation.

Call Format

```
ELN$ALLOCATE_STACK(
    stack_size,
    status
    )
```

Arguments

stack_size. This argument is an INTEGER value that supplies the number of bytes of stack space required.

status. This optional argument is an INTEGER variable that receives the completion status.

DEALLOCATE_STACK

The DEALLOCATE_STACK procedure trims the stack back by the amount specified, but not beyond the page containing the current stack pointer (SP). Specifying an overly large number will cause the stack to be trimmed to the currently needed size. To use the procedure, you must include the module $STACK_UTILITY in the compilation.

Call Format

```
ELN$DEALLOCATE_STACK(
    stack_size,
    status
    )
```

Arguments

stack_size. This argument is an INTEGER value that supplies the number of bytes by which to reduce the current stack size.

status. This optional argument is an INTEGER variable that receives the completion status.

Chapter 13
Errors and Exception Handling

The term *error* in this manual means a violation of the language rules (including violations in calling kernel procedures). The Pascal language was designed so that most errors can be detected by the compiler, without executing the program. Of the errors not detected by the compiler, many can be detected during program execution, but this usually requires additional code.

This chapter discusses errors and the EXCEPTION_HANDLER function type. Exception names and status values are discussed and the exception handling procedures are described in detail. A brief description of each procedure is given, followed by the VAXELN Pascal call format, arguments, and status values (for kernel procedures).

Errors

The following list classifies most error detection in the development system. Not included here is the manner in which the kernel detects and reports errors that were not detected by the compiler. Errors detected by the kernel are reported to the caller of the kernel procedure via a status value returned in the argument list or via an exception. The types of errors are:

- Any violation of a language rule that is not described by the other items in this list is detected by the compiler. Usually, the result is an error-level message, although a warning-level message is issued in a few cases.

- A *warning-level error* is a potential error detected by the compiler, It results only in a warning-level message.
- A *run-time error* is an error that causes an exception if the statement containing the error is executed.
- A *range violation* is, in most cases, an error related to the actual value of a variable or expression, such as an array index outside the limits of the corresponding index range or an ordinal value outside the target's subrange in an assignment. If the values in question can be determined by the compiler, the range violation is detected at compile time. Otherwise, the compiler generates code to check for the violation at run time, but only if the compiler command's CHECK=RANGE qualifier is used. If this qualifier is not used and a range violation occurs during program execution, further effects are unpredictable.
- An *unpredictable error* is an error that is not explicitly detected. If a statement contains an unpredictable error, the compiler may detect it or may detect a related error; however, the error may raise a run-time exception or cause later misbehavior in the program's execution.

Compiler Error Detection

The compiler does not try to detect all errors in a program. Instead, it favors producing only one message for each error, possibly at the expense of masking other, nearby errors. The aim is quick diagnosis, editing, and recompilation of the program.

In many cases, the compiler completely evaluates expressions with constant operands. Such evaluation

may let the compiler detect a run-time error or range violation, either in the evaluated expression or in the statement containing it. In such a case, the compiler usually (but not always) issues a warning-level message and generates code that will cause an exception if executed. The aim is to tolerate errors in statements that will not actually be executed. Because execution of the statement is definitely an error, the compiler may trim its code down to one instruction that raises the relevant exception.

Warning-Level Errors

In general, the compiler issues warning-level messages for things that are likely to be mistakes but which can, in fact, occur in a correct program. Examples include:

- Some trivial syntax errors, such as "; ELSE"
- Some technically correct constructions that are likely to be mistakes, such as "DO ;"
- Cases that are range violations or run-time errors; these may be all right if the statement containing the error is never executed.
- Errors explicitly classified as warning-level errors

The generation of warning-level messages is controlled by a compiler command option. The best practice is to enable warning-level messages and, if feasible, modify the program to eliminate them.

EXCEPTION_HANDLER Function Type

Functions of type EXCEPTION_HANDLER are user-defined handlers for exception conditions. Within a program, procedure, function, or process block, only one function of this type can be established at a time as the exception handler for that block. The established

handler is then called on the occurrence of all exceptions in that block's activation, whether as a result of an operation in the block, as the result of calling the RAISE_EXCEPTION kernel procedure, or as the result of an asynchronous event represented by such exceptions as KER$_QUIT_SIGNAL and KER$_POWER_SIGNAL.

The asychronous exception KER$_POWER_SIGNAL is delivered to programs only if you enable it with the System Builder.

If there is no handler established for the current block, the stack of active blocks is searched for an exception handler. Unhandled exceptions are fatal and cause deletion of the process, unless a debugger is included in the system. If a second exception occurs during the execution of an exception handler, the search for the second exception's handler skips the stack frames that were searched to find the first one. This means that exception handlers are not reentered recursively.

All functions of this type have the result type BOOLEAN. You write a handler to examine the argument list and determine which exception occurred. If the exception is nonfatal (that is, if you can repair the problem dynamically), you return TRUE. If you cannot handle the exception, return FALSE; this causes the exception to be "resignaled," and the kernel attempts to find a handler in an enclosing block.

You can also "unwind" the call stack by using a GOTO statement whose target is a label in a higher-level block. In this case, call frames are removed from the stack, beginning with the exception handler's frame, until the target block is reached. If, when a frame is going to be removed, it has an associated exception handler, that handler is called with the exception name SS$_UNWIND. This exception name indicates to the

handler that it is about to be removed from the stack and gives it a chance to clean up any local variables or other state information.

Exception Arguments and Types

The function type EXCEPTION_HANDLER is used as if the following declaration is present:

```
FUNCTION EXCEPTION_HANDLER(
    VAR SIGNAL_ARGS:
      CHF$R_SIGNAL_ARGS;
    VAR MECH_ARGS:
      CHF$R_MECH_ARGS
    ) : BOOLEAN;
    FUNCTION_TYPE;
```

The CHF$ data types are provided for use in functions of this type; they define the signal and mechanism arguments delivered for particular exceptions.

Signal Arguments

```
TYPE
    CHF$R_SIGNAL_ARGS = RECORD
    { Signal array. }
      ARG_COUNT : INTEGER;
      NAME : INTEGER;
      ADDITIONAL : ARRAY[1..250] OF INTEGER;
    END;
```

This type represents the number and name of the signal arguments, plus an array of additional arguments, if any. The exception names in the system are represented as named integer constants declared in RTLOBJECT.OLB. The additional arguments can be typecast as shown in the examples below.

Mechanism Arguments

```
TYPE
   CHF$R_MECH_ARGS = RECORD
   { Mechanism array. }
     ARG_COUNT : INTEGER;
     FRAME : ↑ ANYTYPE;
     DEPTH : INTEGER;
     SAVR0 : INTEGER;
     SAVR1 : INTEGER;
   END;
```

This type represents the number of mechanism arguments, plus the arguments themselves (frame pointer, frame depth, and saved contents of registers R0 and R1, respectively).

Additional Arguments

```
TYPE
   CHF$R_SIGNAL_ARGS_ADDITIONAL
     (ARG_COUNT : INTEGER) = RECORD
       ARG_ARRAY :
         ARRAY[1..ARGCOUNT - 3]
         OF INTEGER;
       PC: INTEGER;
       PSL : INTEGER;
     END;
```

This flexible type represents additional arguments plus the exception program counter (PC) and processor status longword (PSL). It can be used in typecasting the additional arguments, with the number of additional signal arguments as its extent value.

Examples

```
FUNCTION eh OF TYPE EXCEPTION_HANDLER;
  VAR i : INTEGER;
  BEGIN
  i : = SIGNAL_ARGS.NAME;
  .
  .
  { Return TRUE or FALSE, as appropriate.}
  END;
```

Here, an exception handler named eh obtains the name of the exception that caused it to be called and takes appropriate action: returning TRUE means that the exception was handled; FALSE causes a search of the stack for another handler or, if there is no other handler, deletion of the process.

```
FUNCTION eh OF TYPE EXCEPTION_HANDLER;
  VAR i,j,k : INTEGER;
  BEGIN
  WITH x AS signal_args.additional::
    chf$r_signal_args_additional
    (signal_args.arg_count )
    DO BEGIN
      i : = x.PC;
      j : = x.PSL;
      k : = x.ARG_ARRAY[1];
    END;
  .
  .
  { Return TRUE or FALSE, as appropriate.}
  END;
```

Here, the handler obtains access to the additional signal arguments, to examine the PC and PSL and the first element of the additional argument array. The signal argument signal_args.additional is typecast to

the flexible type representing such arguments, with the number of signal arguments supplied as the extent value.

Related Documentation

For a general discussion of the techniques for writing exception handlers, see the *VAX/VMS Run-Time Library Routines Reference Manual.*

Exception Names and Status Values

The names of most exceptions are listed in Appendix C of the *VAXELN User's Guide.*

The module $PASCALMSG defines the names of exceptions detected by the Pascal run-time routines. $ELNMSG defines the exceptions detected by the compiler and VAXELN run-time routines. Include these modules (from the library RTLOBJECT.OLB) in the compilation of your source program. Note that to be used in exception handlers, the SS$ exception names must be declared in your program with the EXTERNAL and VALUE attributes.

Kernel procedures raise exceptions if they are unsuccessful and you do not request the completion status by using the status parameter. Such exceptions have the same names as the corresponding status values. (That is, KER$_NO_SUCH_PROGRAM can be either a status value or exception name depending on whether you request the status.) The idea is that you can decide *not* to check the status after every call and can instead take an exception in the event of an error.

The status values are defined in the module $KERNELMSG in RTLOBJECT.OLB; include this module in the compilation of your program to use the values for checking completion status.

Exception Handling Procedures

The procedures described in this section relate to VAXELN exception handling and accessing the message data base. Table 13-1 summarizes these procedures.

Table 13-1. Exception Handling Procedures

Procedure	Purpose
ASSERT	checks the validity of a Boolean expression and raises an exception if the result is FALSE.
DISABLE_ASYNCH_EXCEPTION	prevents the delivery of asynchronous exceptions.
ENABLE_ASYNCH_EXCEPTION	allows the delivery of asynchronous exceptions.
ESTABLISH	establishes a function as a block's exception handler.
GET_STATUS_TEXT	returns the text associated with a status code.

Table 13-1. Continued

Procedure	Purpose
RAISE_EXCEPTION	causes a software exception in the calling process.
RAISE_PROCESS_EXCEPTION	raises the KER$_PROCESS_ATTENTION exception.
REVERT	disables exception handling in the current block.
UNWIND	unwinds the call stack to a new location.

Note: To use GET_STATUS_TEXT, include the module $GET_MESSAGE_TEXT from the RTLOBJECT library in the compilation of your program. To use RAISE_PROCESS_EXCEPTION or UNWIND, include the module $KERNEL from the RTLOBJECT library in the compilation of your program.

ASSERT

The ASSERT procedure checks the validity of a BOOLEAN expression. It can be used to validate the arguments received by a function or procedure, for example.

Call Format

```
ASSERT(expression)
```

Arguments

expression. The argument supplies a relational or other BOOLEAN-valued expression.

Notes

If assertion checking is enabled by the command qualifier CHECK=ASSERT, the expression is evaluated by the compiler if possible.

If the compiler evaluation is possible and results in TRUE, the ASSERT procedure has no effect. If the result is FALSE, the compiler issues a warning message and generates code that will raise an exception if executed.

If the expression cannot be evaluated by the compiler, the compiler generates code to check the assertion at run time. If found FALSE at run time, the ELN$ASSERT exception is raised.

If the CHECK qualifier is not used on the compiler command, the ASSERT procedure has no effect, although its argument is still checked for linguistic validity.

DISABLE_ASYNCH_EXCEPTION

The DISABLE_ASYNCH_EXCEPTION kernel procedure prevents the delivery of asynchronous exceptions (such as KER$_QUIT_SIGNAL and KER$_POWER_SIGNAL) to the calling process.

Call Format

```
DISABLE_ASYNCH_EXCEPTION(STATUS : = stat)
```

Arguments

stat. This optional argument is an INTEGER variable that receives the completion status of DISABLE_ASYNCH_EXCEPTION. KER$_SUCCESS is the only possible status.

ENABLE_ASYNCH_EXCEPTION

The ENABLE_ASYNCH_EXCEPTION kernel procedure allows the delivery of asynchronous exceptions (such as KER$_QUIT_SIGNAL and KER$_POWER_SIGNAL) to the calling process. Asynchronous exceptions are enabled by default and must be reenabled only after being explicitly disabled.

Call Format

```
ENABLE_ASYNCH_EXCEPTION(STATUS: = stat)
```

Arguments

stat. This optional argument is an INTEGER variable that receives the completion status of ENABLE_ASYNCH_EXCEPTION. KER$_SUCCESS is the only possible status.

ESTABLISH

The ESTABLISH procedure establishes a specified function as a block's exception handler.

Call Format

```
ESTABLISH(function)
```

Arguments

function. This argument supplies a function of type EXCEPTION_HANDLER.

GET_STATUS_TEXT

The GET_STATUS_TEXT procedure returns the text associated with a status code that you provide as input to the routine. In addition, a format-control parameter can be provided so that the returned string contains only a part of the information available. To use the procedure, you must include the module $GET_MESSAGE_TEXT in the compilation. In addition, you normally link selected object modules with your program to provide the message data base. (For complete information, see the *VAXELN User's Guide.)*

Call Format

```
ELN$GET_STATUS_TEXT(
   msgid ,
   flags,
   result_string
   )
```

Arguments

msgid. This argument is an integer supplying the status code.

flags. This argument is a set that provides format control of the resulting string. (Its type is GET_STATUS_FLAGS, which is also defined in the module $GET_MESSAGE_TEXT.) The result string has four fields, delimited here by angle brackets:

```
% <facility>-<severity>-<msg_ident>, <message_text>
```

The flags argument specifies which of these fields are returned. Specifying an empty set causes all fields to be returned. The following type declarations are provided:

```
TYPE
{ Define the input parameter to control the format of
the text message. }
  get_status_fields  = (status$text,
            status$ident,
            status$severity,
            status$facility);
  get_status_flags  = SET OF get_status_fields;

{ Define a record of fields to overlay the INTEGER
status parameter. }
  status_value_format = PACKED RECORD
        severity : (status$warning,
            status$success,
            status$error,
            status$information,
            status$fatal,
            status$reserved_5,
            status$reserved_6,
            status$reserved_7);
        status_id : 0..%x1fffffff;
        END;
```

For instance, if the argument is

```
[status$severity,status$text]
```

the message will be formatted as follows:

```
%severity, message text
```

result_string. This string variable (type VARYING_STRING(255)) receives the text corresponding to the status code, formatted according to the flags parameter. This procedure always returns text in the result_string even if the status code is not found. In that case, result_string will contain:

```
%facility-severity-NOMSG, Message number <number>
```

For example,

```
ELN$GET_STATUS_TEXT(
ker$_bad_count,
[status$facility,status$severity,status$ident,status$text],
output_string
);
```

will cause output_string to receive:

```
%KERNEL-F-BAD_COUNT, Bad parameter count
```

RAISE_EXCEPTION

The RAISE_EXCEPTION kernel procedure causes a software exception in the calling process.

Call Format

```
RAISE_EXCEPTION(
   exception_name,
   additional_argument-list,
   STATUS : = stat
   )
```

Arguments

exception_name. This argument supplies an integer value denoting a particular exception. Usually, named constants are used. Note that some exception names, such as SS$_ACCVIO, are used to identify specific system or hardware events (in this case, an access violation); take care not to raise one of these exceptions.

additional_argument-list. This list supplies zero or more additional exception arguments that will be made available to the exception handler in the array of additional arguments.

stat. This optional argument is an INTEGER variable that receives the completion status of RAISE_EXCEPTION.

Status Values

KER$_SUCCESS. The procedure completed successfully.

KER$_BAD_COUNT. The procedure call specified an incorrect number of arguments.

KER$_BAD_STACK. The stack size was insufficient.

KER$_NO_ACCESS. An argument specified is not accessible to the calling program.

RAISE_PROCESS_EXCEPTION

The RAISE_PROCESS_EXCEPTION kernel procedure raises the asynchronous exception KER$_PROCESS_ATTENTION in the specified process. To use the procedure, you must include the module $KERNEL in the compilation.

Call Format

```
KER$RAISE_PROCESS_EXCEPTION(
   status,
   process_var
   )
```

Arguments

status. This optional argument is an INTEGER variable that receives the completion status of RAISE_PROCESS_EXCEPTION.

process_var. This argument specifies the process in which the exception is to be raised.

Status Values

KER$_SUCCESS. The procedure completed successfully.

KER$_BAD_TYPE. The process_var argument is not of type PROCESS.

KER$_BAD_VALUE. The process_var argument is invalid or refers to a deleted process.

REVERT

The REVERT procedure disables exception handlers. The effect on program logic is as if no exception handler had been established in the current block.

Call Format

```
REVERT
```

Arguments

There are no arguments.

UNWIND

The UNWIND kernel procedure unwinds the call stack to a new location. To use the procedure, you must include the module $KERNEL in the compilation.

Call Format

```
KER$UNWIND(
    status,
    new_fp,
    new_pc
    )
```

Arguments

status. This optional argument is an INTEGER variable that receives the completion status of UNWIND.

new_fp. This argument supplies the target frame pointer (FP).

new_pc. This argument supplies the new program counter (PC) at the new FP.

Status Values

KER$_SUCCESS. The procedure completed successfully.

KER$_BAD_COUNT. The procedure call specified an incorrect number of arguments.

KER$_NO_ACCESS. An argument specified is not accessible to the calling program.

Chapter 14
Device Drivers and Interrupts

This chapter briefly discusses device driver programs and the kernel services that relate to devices. In addition, interrupt service routines are discussed, including declarations, the handling of device interrupts, recovery from power failure, and the procedures used to manipulate interrupt priority levels.

This chapter also discusses the procedures relating to direct memory access (DMA) UNIBUS and QBUS devices and those relating to device registers. Finally, the utility procedures relating to real-time devices are described in detail.

Device Driver Programs

Device drivers are run by their own jobs, like the jobs running application programs. Typically, a device driver job runs in kernel mode at a higher priority than jobs running application programs.

Interrupt service routines are used to service device interrupts and to handle power recovery in device driver programs. With the CREATE_DEVICE kernel service, you can connect a device interrupt to an interrupt service routine. When connected, the interrupt service routine can take any appropriate action to service the interrupt.

The interrupt service routine can signal the DEVICE object with the SIGNAL_DEVICE kernel service. Only one process continues as a result of a call to SIGNAL_DEVICE.

A DEVICE object is also associated with a description of the physical device. Device descriptions consist of a name for the device, its bus-request priority, and the addresses of the device's interrupt vector and control/status register. They are entered in the system, once, with the System Builder, and are then used by programs (device drivers) via the DEVICE object.

Examples

DEVICE objects and their associated procedures can be used to write drivers for devices that have one unit per controller or multiple units per controller.

Single-Unit Example

This example reads single characters from a hypothetical device controller named ANALOG, which interrupts each time it receives a new input character. ANALOG has only one unit.

```
MODULE analogdriver;
TYPE
   csr_def = PACKED RECORD ...;
   { Definition of control status register. }
   region_def = PACKED RECORD
     dat: CHAR;
     END;
     { Communication region for single characters
     (bytes). }
   csr_ptr = ↑csr_def;
   region_ptr = ↑region_def;
   { Pointer types for CREATE_DEVICE and interrupt
   service routine arguments. }
INTERRUPT_SERVICE isr(intcsr: csr_ptr;
   intreg: region_ptr);
```

```
  BEGIN
    { On occurrence of interrupt, read a single
    character from the control/status register, and
    then signal the DEVICE object: }
    intreg ↑ .dat : =
      READ_REGISTER(intcsr ↑ .data);
    SIGNAL_DEVICE;
  END;
  { End of interrupt service routine. }
PROGRAM analog;
  VAR
    dev: DEVICE;
    devicename: VARYING_STRING(30);
    csr: csr_ptr;
    reg: region_ptr;
    pri: INTEGER;
    { CREATE_DEVICE arguments. }
  BEGIN    { Main program. }
    { Obtain device name from argument list. }
    devicename : = PROGRAM_ARGUMENT(1);
    .
    .
    { Create the DEVICE object. }
    CREATE_DEVICE(
      devicename,
      dev,
      SERVICE_ROUTINE : = isr,
      REGION : = reg,
      REGISTERS : = csr,
      PRIORITY : = pri
      );
    { Disable interrupts while initializing the device.
    }
    DISABLE_INTERRUPT(pri);
```

```
      WRITE_REGISTER(csr↑.csr0, init := true);
      ENABLE_INTERRUPT;
      .
      .
      { Wait for the DEVICE object; when the wait is
      satisfied (because the interrupt service
      procedure calls SIGNAL_DEVICE), a single
      character has been read from the device. }
    WAIT_ANY(dev);
    { Perform any appropriate processing on the new
    character in reg↑.data: }
    .
    .
    END.   { End of program. }
  END; { End of module analogdriver. }
```

Multiple-Unit Example

The following example is for a device named ANALOG, a controller with four units attached. The main program creates a subprocess to handle each unit.

```
  MODULE analogdriver;
  TYPE
    csr_def = PACKED RECORD ...;
    region_def = PACKED RECORD
      dat: ARRAY[0..3] OF CHAR;
      END;
    csr_ptr = ↑csr_def;
    region_ptr = ↑region_def;
    { Types for CREATE_DEVICE arguments. }
  VAR
    reg: region_ptr;
    dev: ARRAY[0..3] OF DEVICE;
    csr: csr_ptr;
```

```
INTERRUPT_SERVICE isr(intcsr: csr_ptr;
  intreg: region_ptr);
  VAR
    unit: INTEGER;
    attn: SET OF 0..3;
  BEGIN
    { Scan the device's attention summary register.
    For each unit with attention set, read the data
    and signal the device object. }
    attn : = READ_REGISTER(intcsr ↑ .attn);
    FOR unit : = 0 TO 3 DO
      BEGIN
        IF unit IN attn THEN
          BEGIN
            intreg ↑ .dat[unit] : =
            READ_REGISTER(intcsr ↑ .data[unit]);
            WRITE_REGISTER(intcsr ↑ .attn,
              attn : = [unit]);
            SIGNAL_DEVICE(
              DEVICE_NUMBER : = unit);
          END   { Done with 1 attention. }
      END       { End of attention scan. }
  END;          { End of interrupt service procedure.
                }
PROGRAM analog;
  VAR
    devicename : VARYING_STRING(30);
    pri: INTEGER;
      { CREATE_DEVICE arguments. }
    uproc: ARRAY[0..3] OF PROCESS;
    unit: INTEGER;
```

```
BEGIN { Main program. }
  devicename : = PROGRAM_ARGUMENT(1);
  .
  .
  { Create a DEVICE value for each unit. }
  CREATE_DEVICE(devicename,
    dev, { Four-element array of DEVICE. }
    SERVICE_ROUTINE : = isr,
    REGION : = reg,
    REGISTERS : = csr,
    PRIORITY : = pri
    );
  { Disable interrupts while initializing device. }
  DISABLE_INTERRUPT(pri);
  WRITE_REGISTER(csr ↑ .csr0, init : = true);
  ENABLE_INTERRUPT;
  { Create a subprocess to handle each unit: }
  FOR unit : = 0 TO 3 DO
    CREATE_PROCESS(
      uproc[unit],
        { PROCESS variable. }
      unit_process,
        { Name of process block. }
      unit
        { Unit number. }
      );
  .
  .
  END. { End of main program. }
PROCESS_BLOCK unit_process(unit: INTEGER);
  { Process to handle information from one unit. }
  BEGIN
    .
    .
    WAIT_ANY(dev[unit]);
```

```
        { When the wait is over, take any appropriate
        action to process the character in
        reg ↑ .dat[unit]. }
        .
        .
    END; { End of process block. }
END; { End of module analogdriver. }
```

The kernel services relating to devices are described in detail in the following section. A brief description of each procedure is given, followed by the VAXELN Pascal call format, arguments, and status values.

Kernel Services for Devices

The kernel services described in this section relate to devices and DEVICE objects. Table 14-1 summarizes these procedures.

Table 14-1. Kernel Services for Devices

Procedure	Purpose
CREATE_DEVICE	establishes a connection between a physical device, a program, and an interrupt service routine.
SIGNAL_DEVICE	signals a DEVICE object from an interrupt service routine.

CREATE_DEVICE

The CREATE_DEVICE procedure establishes a connection between a physical device, a program, and an interrupt service routine. It creates one or more objects of type DEVICE, which are used to synchronize the program with the device. CREATE_DEVICE can be called only from a program running in kernel mode.

Call Format

```
CREATE_DEVICE(
   device_name,
   device,
   VECTOR_NUMBER : = relative_vector,
   SERVICE_ROUTINE : = routine_name,
   REGION : = region_pointer,
   REGISTERS : = register_pointer,
   ADAPTER_REGISTERS : = adapter_pointer,
   VECTOR : = vector_pointer,
   PRIORITY : = interrupt_priority,
   POWERFAIL_ROUTINE : = power_routine,
   STATUS: = stat
   )
```

Arguments

device_name. This argument supplies a 1–30-character string naming the device. The name must match one of the device names established with the System Builder.

device. This argument is a DEVICE variable that receives the identifier of the new DEVICE object or objects. It can be a single DEVICE variable or an array of 1 to 16 DEVICE elements, the lower bound of which must be 0. If you specify an array, a DEVICE object is created for and its identifier is placed in each element.

relative_vector. This optional argument supplies an integer from 1 to 128, specifying which vector of a multiple-interrupt-vector device should be connected to the interrupt service routine. (The base vector address is part of the device description established with the System Builder.) If this argument is omitted, the default is 1 (first vector).

routine_name. This optional argument supplies the name of an interrupt service routine. It can be omitted, to drive a device by polling instead of with interrupts. If its name is supplied, the routine is called by the kernel on the occurrence of a device interrupt.

region_pointer. This optional argument is a variable of any appropriate pointer type (except ↑ ANYTYPE) and receives a pointer to the communication region of the interrupt service routine. CREATE_DEVICE uses the size of the pointer variable's base type to establish a region of the appropriate size; the region is zeroed by CREATE_DEVICE. The pointer is also passed by the kernel to the interrupt service routine on the occurrence of a device interrupt. If the argument is omitted, no region is created, and the interrupt service routine (if any) receives NIL instead of the region's address. Note that every call with this parameter creates a new communication region; if you use the same pointer variable from one call to another, the procedure will overwrite its previous value with the address of the new communication region.

register_pointer. This optional argument is a variable of an appropriate pointer type that receives a pointer to the first device control register. (The I/O space address of the first control register is part of the device description established with the System Builder.) The pointer is also passed to the interrupt service routine on the occurrence of a device interrupt. This argument can

be omitted. Within the interrupt service routine, the corresponding parameter is declared to specify the type of the register pointer.

adapter_pointer. This optional argument is a variable of an appropriate pointer type that receives a pointer to the first adapter (UNIBUS or QBUS) control register.

vector_pointer. This optional argument is a variable of an appropriate pointer type that receives a pointer to the interrupt vector in the system control block. The interrupt vector address is part of the device description established with the System Builder.

interrupt_priority. This optional argument is an INTEGER variable that receives the interrupt priority level (IPL) of the device. The bus-request interrupt priority level is part of the device description established with the System Builder.

power_routine. This optional argument supplies the name of an interrupt service routine that is called, before any process or interrupt service routine is restarted, when the processor enters a power recovery sequence.

stat. This optional argument is an INTEGER variable that receives the completion status of CREATE_DEVICE.

Status Values

KER$_SUCCESS. The procedure completed successfully.

KER$_BAD_MODE. The procedure was called from a program that was not running in kernel mode; kernel mode is required for this procedure.

KER$_BAD_VALUE. The device argument is an array with more than 16 elements.

KER$_DEVICE_CONNECTED. The device named in the CREATE_DEVICE call is already connected to a DEVICE value.

KER$_NO_ACCESS. An argument specified is not accessible to the calling program.

KER$_NO_OBJECT. No free job object table entries are currently available. There are a maximum of 1024 object table entries per job.

KER$_NO_POOL. No free system pool is currently available. The size of the system pool can be set by the System Builder utility.

KER$_NO_SUCH_DEVICE. The device name specified in a CREATE_DEVICE call cannot be found in the list of devices created by the System Builder utility.

KER$_NO_SYSTEM_PAGE. No free system page table entries are currently available to map the I/O region.

SIGNAL_DEVICE

The SIGNAL_DEVICE procedure signals a DEVICE object from an interrupt service routine. It can be called only from an interrupt service routine or a subroutine thereof.

Call Format

```
SIGNAL_DEVICE(
   DEVICE_NUMBER : = integer,
   STATUS: = stat
   )
```

Arguments

integer. This optional argument supplies an integer in the range 0–15, identifying the element in a DEVICE array to be signaled.

stat. This optional argument is an INTEGER variable that receives the completion status of SIGNAL_DEVICE.

Note: No exceptions are raised by the procedure, even if status is not requested and an error occurs.

Status Values

KER$_SUCCESS. The procedure completed successfully.

KER$_BAD_VALUE. The integer argument is out of range.

Interrupt Service Routine Declarations

An interrupt service routine resembles a procedure with two pointer parameters. The routine is called by the kernel directly on the occurrence of a device interrupt, and the parameters then point to the first control register of a device and to a communication region.

The syntax of an interrupt service routine declaration is shown in Figure 14-1.

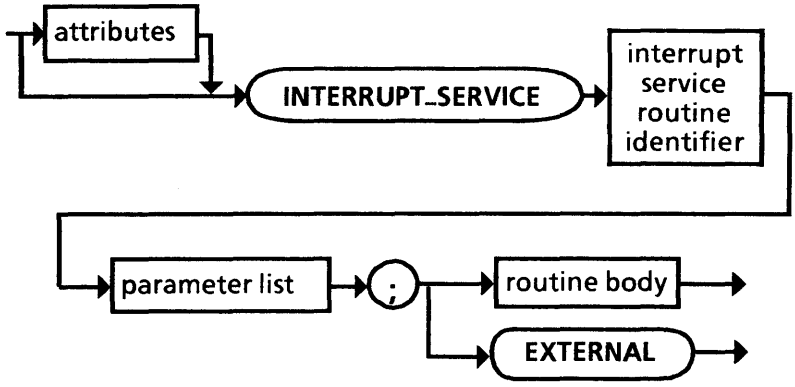


Figure 14-1. Interrupt Service Routine Declaration Syntax

The interrupt service routine identifier is declared at the outer level as the name of the interrupt service routine. This declaration is only allowed at the outer level. The declaration cannot be used as a complete compilation unit; it must occur within the MODULE ... END construction.

The routine body gives the local declarations and executable code for the interrupt service routine. One of the attributes UNDERFLOW or NOUNDERFLOW may be specified; it applies to the routine body, as explained in Chapter 2, "Program Structure."

The EXTERNAL directive may be used to indicate that the routine body is coded in another programming language. In this case, neither UNDERFLOW nor NOUNDERFLOW is allowed.

The parameter list has the same form as in a procedure or function declaration. However, it must declare exactly two parameters: both value parameters, and both having pointer data types. The first parameter is a pointer to the device's first control/status register. The second parameter is a pointer to the communication region defined by the call to CREATE_DEVICE that established the interrupt service routine.

The interrupt service routine's body is executed in a special context in which the only memory accessible is system space (where the communication region is mapped) and the local storage of the block. Thus, it has access only to its local variables, the communication region, and the device registers that are addressed by the first parameter of the interrupt service routine.

In particular, an interrupt service routine must not access an outer-level variable unless it has the VALUE or READONLY attribute. The compiler detects this error for the interrupt service routine's body and any routines declared within it. However, any other access to improper data (for example, via pointers or by some other routine called by the interrupt service routine) is an unpredictable error.

Note that pointers to other communication regions can be stored in a communication region, for use with multivector devices.

Usually, few routines are called from an interrupt service routine, and some cannot be called. Specifically excluded are all kernel procedures except SIGNAL_DEVICE, the predeclared I/O procedures (CLOSE, OPEN, READ, PUT, and so forth), and NEW and DISPOSE.

Interrupt Handling

When a device's interrupts are connected to an interrupt service routine (by the CREATE_DEVICE procedure), the specified routine is called by the kernel each time the device interrupts the processor.

The interrupt service routine can take any action needed to service the interrupt, using the device register pointer to gain access to the device registers. Typically, with devices that interrupt for several reasons, the interrupt service routine can examine the control status register to determine which interrupt has occurred.

The interrupt service routine can communicate with the main program via the communication region. In fact, the communication region supplies any and all data available in the interrupt service routine other than its own local variables and outer-level constants.

The routine also can call SIGNAL_DEVICE to synchronize the main program with the interrupt. (The main program waits for the handling of an interrupt by using WAIT_ANY or WAIT_ALL to wait for the associated DEVICE object to be signaled.)

A typical declaration begins this way:

```
INTERRUPT_SERVICE isr (
    register : ↑ mycsr;
    comm :  ↑ myregion
    );
BEGIN{Actions to service interrupt. }
    IF comm ↑ .count = ...;
    .
    .
END;
```

The name of this routine (isr) is then used as the argument for the SERVICE_ROUTINE parameter of CREATE_DEVICE.

Power-Recovery Handling

Devices normally need special attention following a power failure, and the necessary speed and synchronization requirements cannot be met by the general power-recovery exception KER$_POWER_SIGNAL. For this reason, you can specify, in a CREATE_DEVICE call, the name of an interrupt service routine that is called when the processor enters its power-recovery sequence. Such a routine is called before any other process or ordinary interrupt service routine is restarted.

The VAX architecture defines a power-failure interrupt at interrupt priority level (IPL) 31. Therefore, a process can set the processor's IPL to 31 and block the interrupt, allowing it to synchronize itself with the power-recovery routine.

More information on interrupt priority levels is provided in the *VAXELN User's Guide*.

The following example illustrates the power-recovery facilities provided by VAXELN:

```
CONST
   IPL$POWER = 31;

TYPE
   myregion = RECORD...END;
   mycsr = ...;

VAR
   com : ↑myregion;
   reg : ↑mycsr;
   dev : DEVICE;
   ipl : INTEGER;

INTERRUPT_SERVICE power (
   regptr : ↑mycsr;
   comm :  ↑myregion);
   BEGIN  { Actions for power recovery. }
     { Reinitialize controller. }
     WRITE_REGISTER(regptr↑.csr,
     master_clear : = TRUE);
     { Set state for other parts of program. }
     comm↑.powerfail = TRUE;
     { If someone was busy, signal the device. }
     IF comm↑.busy THEN SIGNAL_DEVICE;
   END;

INTERRUPT_SERVICE isr (
   regptr : ↑mycsr;
   comm : ↑ myregion);
   BEGIN
     comm↑.busy : = FALSE; { Clear state. }
     .
     .
     SIGNAL_DEVICE;
   END;
```

```
BEGIN { Device driver program. }
  .
  .
  CREATE_DEVICE('MYDEV',
    dev,
    SERVICE_ROUTINE : = isr,
    REGION : = com,
    REGISTERS : = reg,
    PRIORITY : = ipl,
    POWERFAIL_ROUTINE : = power);
  .
  .
  REPEAT
    com ↑ .powerfail : = FALSE;
    DISABLE_INTERRUPT(ipl);
    WRITE_REGISTER(reg ↑ .bytecount, bufsize);
    DISABLE_INTERRUPT(IPL$POWER);
    IF NOT com ↑ .powerfail
    THEN
      BEGIN
      WRITE_REGISTER(reg ↑ .csr,
        function : = read,
        go : = TRUE,
        intenable : = TRUE);
      com ↑ .busy : = TRUE;
      ENABLE_INTERRUPT;
      WAIT_ANY(dev);
      END
    ELSE
      ENABLE_INTERRUPT;
  UNTIL NOT com ↑ .powerfail;
  .
  .
  END.
```

For more examples, see the Pascal sources for the drivers delivered with your development system.

IPL Procedures

The procedures described in this section raise or lower the processor's interrupt priority levels. Table 14-2 summarizes these procedures.

Table 14-2. IPL Procedures

Procedure	Purpose
DISABLE_INTERRUPT	prevents interrupts from a device by raising the interrupt priority level.
ENABLE_INTERRUPT	allows interrupts from a device by lowering the interrupt priority level to 0.

DISABLE_INTERRUPT

The DISABLE_INTERRUPT procedure prevents interrupts from a device, by raising the interrupt priority level (IPL) of the processor to the IPL of the device. It can be called only from programs running in kernel mode. While interrupts are disabled, no kernel procedures can be called; attempting to do so causes unpredictable results.

Call Format

DISABLE_INTERRUPT(priority)

Arguments

priority. This argument supplies an integer in the range 1–31, giving the new interrupt priority level.

Notes

The current interrupt priority level is part of the processor-wide state of a VAX computer. Disabling interrupts of a certain priority also disables all other system activities that occur at or below that priority level. In essence, if the IPL is raised by a process to block device interrupts, that process is the only activity, other than interrupt service routines, that can execute until the process lowers the IPL by calling ENABLE_INTERRUPT.

If the power fails while interrupts are disabled, the IPL is set to zero before the KER$POWER_SIGNAL exception is raised. This exception is handled like any other, synchronous exception, but continuing from the exception if it occurs with interrupts disabled has unpredictable effects.

ENABLE_INTERRUPT

The ENABLE_INTERRUPT procedure allows interrupts from a device by lowering the interrupt priority level (IPL) of the calling process to minimum priority (0). It can be called only from programs running in kernel mode.

Call Format

```
ENABLE_INTERRUPT
```

Arguments

There are no arguments.

DMA Device Handling Procedures

The procedures described in this section are used in device driver programs for UNIBUS and QBUS direct memory access (DMA) devices. Table 14-3 summarizes these procedures.

Table 14-3. DMA Device Handling Procedures

Procedure	Purpose
ALLOCATE_MAP	allocates a contiguous block of UNIBUS or QBUS map registers.
ALLOCATE_PATH	allocates a UNIBUS adapter buffered datapath.
FREE_MAP	frees previously allocated UNIBUS or QBUS map registers.
FREE_PATH	frees a previously allocated UNIBUS adapter buffered datapath.
LOAD_UNIBUS_MAP	loads UNIBUS or QBUS map registers.
PHYSICAL_ADDRESS	returns the physical address of the variable supplied as its argument.
UNIBUS_MAP	maps memory buffers for direct memory access by UNIBUS or QBUS devices.

Table 14-3. Continued

Procedure	Purpose
UNIBUS_UNMAP	unmaps memory buffers previously mapped for direct memory access by a UNIBUS or QBUS device.

Notes: To use ALLOCATE_MAP, ALLOCATE_PATH, FREE_MAP, or FREE_PATH, include the module $KERNEL from the RTLOBJECT library in the compilation of your program.

To use LOAD_UNIBUS_MAP, UNIBUS_MAP, or UNIBUS_UNMAP, include the module $UNIBUS from the RTLOBJECT library in the compilation of your program.

To use PHYSICAL_ADDRESS, include the module $PHYSICAL_ADDRESS from the RTLOBJECT library in the compilation of your program.

ALLOCATE_MAP

The ALLOCATE_MAP procedure allocates a contiguous block of UNIBUS or QBUS map registers for use by a device driver program to map VAX memory to UNIBUS or QBUS memory addresses. It can be called only from programs running in kernel mode. To use the procedure, you must include the module $KERNEL in the compilation.

Call Format

```
KER$ALLOCATE_MAP(
   status,
   register,
   number,
   count,
   device_object,
   spt_address
   )
```

Arguments

status. This optional argument is an INTEGER variable that receives the completion status of ALLOCATE_MAP.

register. This argument is a pointer variable that receives a pointer (↑ ANYTYPE) to the first register allocated.

number. This argument is an INTEGER variable that receives the starting map register number (0–495).

count. This argument supplies an INTEGER expression giving the number of registers to allocate.

device_object. This argument supplies the DEVICE value that identifies the device for which the registers are to be used.

spt_address. This optional argument is a pointer variable that receives the base address (↑ ANYTYPE) of the system page table (SPT).

Status Values

KER$_SUCCESS. The procedure completed successfully.

KER$_BAD_MODE. The procedure was called from a program that was not running in kernel mode; kernel mode is required for this procedure.

KER$_BAD_TYPE. The device_object argument is not of type DEVICE.

KER$_BAD_VALUE. The device_object argument is invalid or refers to a deleted device.

KER$_NO_ACCESS. An argument specified is not accessible to the calling program.

KER$_NO_MAP_REGISTER. No free map registers are currently available. There are 496 map registers per UNIBUS or QBUS.

ALLOCATE_PATH

The ALLOCATE_PATH procedure allocates a UNIBUS adapter buffered datapath for use by a direct memory access (DMA) UNIBUS device. It can be called only from programs running in kernel mode. To use the procedure, you must include the module $KERNEL in the compilation.

Call Format

```
KER$ALLOCATE_PATH(
   status,
   register,
   number,
   dev
   )
```

Arguments

status. This optional argument is an INTEGER variable that receives the completion status of ALLOCATE_PATH.

register. This argument is a pointer variable that receives a pointer (↑ANYTYPE) to the allocated datapath register.

number. This argument is a INTEGER variable that receives the allocated datapath register number (1..3).

dev. This argument supplies the DEVICE value that identifies the device for which the datapath is allocated.

Notes

A buffered datapath can be used to optimize the use of memory by a DMA device that does strictly sequential address transfers. (For additional information on

buffered datapaths, see the *VAX Hardware Handbook.*) The VAX-11/750 is the only processor supported by VAXELN that has UNIBUS buffered datapaths.

To use a buffered datapath for a DMA transfer, the allocated datapath number must be loaded into the UNIBUS map registers being used for the transfer. The UNIBUS_MAP and LOAD_UNIBUS_MAP procedures accept an optional datapath number for loading into the UNIBUS map registers.

When a UNIBUS buffered datapath is used for a DMA transfer, the datapath must be "purged" when the transfer has completed. This is accomplished by writing a value of 1 to the datapath register, identified by the returned register pointer.

Status Values

KER$_SUCCESS. The procedure completed successfully.

KER$_BAD_MODE. The procedure was called from a program that was not running in kernel mode; kernel mode is required for this procedure.

KER$_BAD_TYPE. The dev argument is not of type DEVICE.

KER$_BAD_VALUE. The dev argument is invalid or refers to a deleted device.

KER$_NO_ACCESS. An argument specified is not accessible to the calling program.

KER$_NO_PATH_REGISTER. No free UNIBUS adapter datapath register is currently available. There are three buffered datapaths per VAX-11/750 UNIBUS adapter.

FREE_MAP

The FREE_MAP procedure frees a set of previously allocated UNIBUS or QBUS map registers. It can be called only from programs running in kernel mode. To use the procedure, you must include the module $KERNEL in the compilation. Any pointers to the freed registers become invalid.

Call Format

```
KER$FREE_MAP(
   status,
   count,
   number,
   device_object
   )
```

Arguments

status. This optional argument is an INTEGER variable that receives the completion status of FREE_MAP.

count. This argument is an INTEGER expression that supplies the number (count) of contiguous map registers to be freed.

number. This argument is an INTEGER expression that supplies the map register number of the first map register, such as the one returned by ALLOCATE_MAP.

device_object. This argument is a DEVICE value that identifies the device for which the registers are freed.

Status Values

KER$_SUCCESS. The procedure completed successfully.

KER$_BAD_MODE. The procedure was called from a program that was not running in kernel mode; kernel mode is required for this procedure.

KER$_BAD_VALUE. The device_object argument is invalid or identifies a deleted device.

KER$_NO_ACCESS. An argument specified is not accessible to the calling program.

FREE_PATH

The FREE_PATH procedure frees a previously allocated UNIBUS adapter buffered datapath. It can be called only from programs running in kernel mode. To use the procedure, you must include the module $KERNEL in the compilation. The VAX–11/750 is the only processor supported by VAXELN that has UNIBUS buffered datapaths.

Call Format

```
KER$FREE_PATH(
   status,
   number,
   dev
   )
```

Arguments

status. This optional argument is an INTEGER variable that receives the completion status of FREE_PATH.

number. This argument is an INTEGER value that supplies the datapath register number, such as the one returned by ALLOCATE_PATH.

dev. This argument supplies the DEVICE value that identifies the device for which the datapath is freed.

Status Values

KER$_SUCCESS. The procedure completed successfully.

KER$_BAD_MODE. The procedure was called from a program that was not running in kernel mode; kernel mode is required for this procedure.

KER$_BAD_VALUE. The dev argument is invalid or identifies a deleted device.

KER$_NO_ACCESS. An argument specified is not accessible to the calling program.

LOAD_UNIBUS_MAP

The LOAD_UNIBUS_MAP procedure is used in device driver programs to load UNIBUS or QBUS map registers for use by a direct memory access UNIBUS or QBUS device. This is an alternative procedure to the more commonly used UNIBUS_MAP procedure. It can be called only from programs running in kernel mode. To use the procedure, include the module $UNIBUS in the compilation.

Call Format

```
ELN$LOAD_UNIBUS_MAP(
    map_register,
    buffer,
    buffer_size,
    spt_address,
    datapath
    )
```

Arguments

map_register. This argument is a pointer to the first UNIBUS or QBUS map register allocated by ALLOCATE_MAP.

buffer. This argument is a variable of type BYTE_DATA(buffer_size), representing the I/O buffer.

buffer_size. This argument is an INTEGER value supplying the buffer size.

spt_address. This optional argument is a pointer to the System Page Table. If this argument is not supplied, a device communication region (or any system space buffer) cannot be mapped.

datapath. This optional argument supplies a UNIBUS datapath to be used for the transfer. If the argument is not supplied, datapath 0, the direct datapath, is used.

Notes

The LOAD_UNIBUS_MAP procedure assumes that sufficient map registers have been allocated by the calling program using the ALLOCATE_MAP procedure (UNIBUS_MAP allocates them for the caller). The LOAD_UNIBUS_MAP procedure also assumes that one additional map register (beyond the number actually necessary to map the buffer) has been allocated for use as an invalid "wild-transfer-stopper."

PHYSICAL_ADDRESS

The PHYSICAL_ADDRESS function returns the physical address of a data item. To use the function, you must include the module $PHYSICAL_ADDRESS in the compilation.

Call Format

RESULT : = PHYSICAL_ADDRESS(*pointer*)

Arguments and Result

pointer. This argument is a pointer value of any type (↑ ANYTYPE) supplying the virtual address of a data item.

The result is an INTEGER value denoting the physical address of the data item.

UNIBUS_MAP

The UNIBUS_MAP procedure is used in device driver programs to map memory buffers for direct memory access by UNIBUS or QBUS devices. That is, the specified buffer is mapped into the UNIBUS or QBUS address space, and the address of the first register is returned. It can be called only from programs running in kernel mode. To use the procedure, you must include the module $UNIBUS in the compilation.

Call Format

```
ELN$UNIBUS_MAP(
    dev,
    buffer,
    buffer_size,
    unibus_address,
    datapath
    )
```

Arguments

dev. This argument is a DEVICE value identifying the device that will use the mapped memory.

buffer. This argument is a variable of type BYTE_DATA(buffer_size), representing an I/O buffer.

buffer_size. This argument is an INTEGER value supplying the buffer size.

unibus_address. This argument is an INTEGER variable that receives the 18-bit UNIBUS address or 22-bit QBUS address of the mapped buffer.

datapath. This optional argument supplies an integer that specifies the UNIBUS adapter datapath to use. The default is 0, specifying the unbuffered data path.

Notes

The UNIBUS_MAP procedure allocates the correct number of map registers by calling ALLOCATE_MAP. It then converts the virtual address of each page of the buffer to a physical address and stores and validates the physical page numbers in the allocated map registers. If a datapath other than 0 is specified, it is also stored in the map registers. Although the map registers are allocated by UNIBUS_MAP before use, a nonzero datapath number is assumed to be unused by any other device.

UNIBUS_UNMAP

The UNIBUS_UNMAP procedure is used in device driver programs to unmap memory buffers previously mapped for direct memory access by a UNIBUS or QBUS device. It can be called only from programs running in kernel mode. To use the procedure, you must include the module $UNIBUS in the compilation.

Call Format

```
ELN$UNIBUS_UNMAP(
    dev,
    buffer,
    buffer_size,
    unibus_address
    )
```

Arguments

dev. This argument is a DEVICE value identifying the device which was using the mapped memory.

buffer. This argument is a variable of type BYTE_DATA(buffer_size), representing an I/O buffer.

buffer_size. This argument is an INTEGER value supplying the buffer size.

unibus_address. This argument is an INTEGER variable supplying the 18-bit UNIBUS address or 22-bit QBUS address of the mapped buffer.

Notes

The UNIBUS_UNMAP procedure deallocates the correct number of map registers by calling FREE_MAP.

Device Register Procedures

The procedures described in this section read and write device registers and internal processor registers. Table 14-4 summarizes these routines.

Table 14-4. Device Register Procedures

Routine	Purpose
MFPR	returns the current contents of a VAX processor register.
MTPR	moves a specified value into a specified VAX internal processor register.
READ_REGISTER	returns the contents of a device register.
WRITE_REGISTER	loads a specified value or group of values into a specified target variable.

MFPR

The MFPR function returns the current contents of a VAX processor register. It can be called only from programs running in kernel mode.

Call Format

```
RESULT : = MFPR(reg_number)
```

Arguments and Result

reg_number. The argument is an integer expression whose value identifies the specific processor register. See the *VAX Hardware Handbook* or other documentation for the target processor you are using.

The result is an INTEGER value representing the register contents.

MTPR

The MTPR procedure moves a specified value into a specified VAX internal processor register. It can be called only from programs running in kernel mode.

Caution: Processor registers are a privileged system resource. Changing the contents of processor registers while a system is running may cause a fatal exception.

Call Format

```
MTPR(
   reg_number,
   expression
   )
```

Arguments

reg_number. This argument supplies an integer identifying a specific register. See the *VAX Hardware Handbook* or other documentation for the processor model in use.

expression. This argument supplies a value to be loaded into the specified register. The value must be an integer, a pointer, or an item with the LONG attribute.

READ_REGISTER

The READ_REGISTER function returns the value of a variable reference. The operation is performed by a single machine instruction and is not affected by any compiler optimizations. This is the only safe method for reading a device register, and it can also be used safely to read a shared variable.

This function should always be used, instead of a direct assignment, to read the fields in a device register. This use is required because the VAX architecture does not permit certain instructions (in particular, the variable-length bit-field instructions) to be used to read device registers. Using READ_REGISTER ensures that the compiler will generate only the allowed instructions.

Note: To read the contents of a VAX internal processor register, see the MFPR function.

Call Format

```
RESULT : = READ_REGISTER(variable)
```

Arguments and Result

variable. The argument is a variable of type INTEGER, a pointer type, or a type with the BYTE, WORD, or LONG attribute.

The result is the current value of the variable and has the same data type.

Use with Actual Device Registers

To model a device register, consult the device's hardware manual for the register definition and then declare a Pascal variable whose internal representation matches the register. Typically, actual device registers are modeled by pointers to integers or records.

The pointer value (that is, the register address) is obtained in the program via the CREATE_DEVICE procedure and the pointer is then used as the READ_REGISTER argument. The register's I/O space address is supplied to the system when you build it with the System Builder.

For example:

```
PROCEDURE x;
   TYPE
   csr = [BYTE] PACKED RECORD
            go: BOOLEAN;
            ready : BOOLEAN;
            END;
   VAR
   csrpointer : ↑csr;
   currentcsr : csr;
      .
      .
```

```
    BEGIN
      .
      .
    CREATE_DEVICE(dev...,
        REGISTERS : = csrpointer,...);
    currentcsr : = READ_REGISTER(csrpointer ↑ );
    IF currentcsr.ready THEN ...;
      .
      .
END;
```

WRITE_REGISTER

The WRITE_REGISTER procedure loads a specified value or group of values into a specified target variable reference. The write operation is performed by a single machine instruction and is not affected by any compiler optimizations. This is the only safe method for writing device registers, and it can also be used to safely write a shared variable.

In one form, the procedure can write into specific bits within the register, modeled as fields in a packed record. In such a case, the procedure can be viewed as building a temporary data item of the necessary size, zeroing all its bits, and then assigning to the specified fields; the resulting temporary value is then used as the source operand of the single VAX MOVE instruction.

This procedure should always be used, instead of a direct assignment statement, to write the fields in a device register. This is required because the VAX architecture does not allow certain instructions (in particular, the variable-length bit-field instructions) to be used to write device registers. Calling WRITE_REGISTER ensures that the compiler generates only the allowed instructions.

Note: To write the contents of a VAX internal processor register, see the MTPR procedure. Note also that this procedure is not restricted to use with actual device registers; it can be used to write the contents of any suitably declared variable.

Call Formats

There are two call formats.

Format 1
```
WRITE_REGISTER(
    target_variable,
    expression
    )
```

Format 2
```
WRITE_REGISTER(
    target_variable,
    field : = expression,
    field : = expression,...
    )
```

Arguments

target_variable. This argument is a reference to a device register or other variable with a suitable data type. In Format 1, it must be an INTEGER variable, a pointer variable, or a variable whose type has the BYTE, WORD, or LONG attribute. In Format 2, it must be a packed record with one of these attributes.

expression. Each expression optionally supplies a value compatible for assignment to the register or field. In Format 2, each field is the name of a field in the indicated packed record, and each must be distinct. The expression with each field must be assignment compatible with the record field. There can be fewer expressions than record fields, but not more. If a record field has no corresponding expression in this list, it is cleared. If the expression is omitted from Format 1, the

entire register is cleared. Any register bits not explicitly declared are cleared by the procedure.

Use with Actual Device Registers

To model a device register, consult the device's hardware manual for the register definition and then declare a Pascal variable whose internal representation matches the register.

The device drivers supplied with your development system contain many programming examples. Typically, actual device registers are modeled by pointers to integers or records. The pointer value (that is, the register address) is obtained in the program via the CREATE_DEVICE procedure and the dereferenced pointer is then used as a WRITE_REGISTER argument. The register's I/O space address is supplied to the system when you build it with the System Builder.

For example, suppose a device register is defined as shown in Figure 14-2:

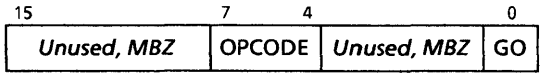


Figure 14-2. Hypothetical Device Register

In this case, GO is a single bit, the low-order bit in the register, and OPCODE (bits 4–7) is a four-bit field that must be loaded with an integer operation code in the range 1–7. All other bits are unused but must be zero (MBZ).

Note that, in general, the OPCODE field may already contain a value from some previous operation.

Therefore, it is not enough merely to "set the appropriate bits" to represent the new operation code, because some bits from the previous value might remain set. Note also that (as is typical of device registers) the OPCODE field is large enough to hold 16 codes, but only 7 are meaningful.

One model of the register is a packed record with a one-bit (BOOLEAN) field at the beginning and a three-bit field beginning at bit 4. The unused fields in the register must be zero, but this is guaranteed by the operation of WRITE_REGISTER, which puts zeros in any positions that are not written explicitly. Bit 7, the meaningless part of the OPCODE field, also is zeroed.

```
TYPE
  dregister: [WORD] PACKED RECORD
    go: [POS(0)] BOOLEAN;
    opcode: [POS(4)] 1..7;
    END;
VAR
  regptr: ↑dregister;
BEGIN
  { Obtain register address and assign to regptr. }
  .
  .
  { Load the register. }
  WRITE_REGISTER(
    regptr↑,
    go: = TRUE,
    opcode: = 4
    );
END.
```

Notice that, because the data type of field opcode is the subrange 1..7, the assignment of some other integer would be a range violation.

Real-Time Device Drivers

The VAXELN development system includes device drivers for the following real-time devices:

- The ADV11C or AXV11C analog-to-digital converter. The AXV11C is the same as an ADV11C, but it also has two digital-to-analog output channels.
- The KWV11C programmable, real-time clock. The KWV11C can be used to initiate action after a specified time interval (via an interrupt or an external signal) or to time an event.
- The DLVJ1 asynchronous serial line controller. The DLVJ1 (formerly DLV11–J) is a QBUS interface that contains four asynchronous serial line channels; it is intended for real-time applications that collect data and control real-time devices using asynchronous serial lines.
- The DRV11–J parallel line interface. The DRV11–J is a QBUS interface that provides communication between a MicroVAX and up to four user devices in 16-bit word lengths via four I/O ports.

The design of these drivers prohibits accessing a given device from more than one job. However, gaining access from different processes within the same job is possible, provided the caller ensures there is no simultaneous access to the same device.

For more information about the real-time devices described above and their associated device drivers, please refer to the *VAXELN User's Guide.*

AXV Device Driver Utility Procedures

The procedures described in this section are used in programs that access ADV11C or AXV11C real-time devices. (The *LSI–11 Analog System User's Guide* contains information on the hardware.) Table 14-5 summarizes these procedures.

Table 14-5. AXV Device Driver Utility Procedures

Procedure	Purpose
AXV_INITIALIZE	readies an ADV or AXV device for input and/or output and creates all needed data structures.
AXV_READ	reads analog data from the specified channels, converts it to binary form, and stores it in a data array.
AXV_WRITE	writes a value to an analog-to-digital conversion output register on an AXV device.

Notes: To use these procedures, include the module $AXV_UTILITY from the RTLOBJECT library in the compilation of your program.

The following Pascal types are declared in module $AXV_UTILITY for use with the AXV device driver utility procedures:

```
TYPE
   { Input/output data for results of conversions. }
   axv$data = -%o3777..%o7777;
```

```
{ Gain data. }
axv$gain = (axv$gain_one,
            axv$gain_two,
            axv$gain_four,
            axv$gain_eight);

{ Identifiers - one for each physical device. }
axv$ = ↑anytype;

{ Data packets passed to axv_read. }
axv$data_array(values_per_chan,
    first_chan,last_chan:integer) = packed
    array[1..values_per_chan,first_chan..last_chan]
    of [word]axv$data;

axv$gain_array(first_chan,last_chan:integer) =
    packed array[first_chan..last_chan] of
    [byte]axv$gain;

{ Dac channel selections for axv_write. }
axv$dac_channel = (axv$dac_channel_a,
    axv$dac_channel_b);
```

AXV_INITIALIZE

The AXV_INITIALIZE procedure readies an ADV or AXV device for input and/or output and creates all needed data structures. It can be called only from programs running in kernel mode. This procedure must be called at least once for each AXV or ADV device used. (The only reason to call this procedure more than once for a single device is to change the values of the BOOLEAN flag parameters or the maximum_values parameter.) To use the procedure, you must include the module $AXV_UTILITY in the compilation.

Call Format

```
ELN$AXV_INITIALIZE(
    device_name,
    identifier,
    maximum_values,
    clock_start_enable,
    external_start_enable,
    re_initialize,
    use_polling,
    status
    )
```

Arguments

device_name. This argument supplies a string of up to 30 characters giving the name of the device to be initialized. This name must match the name established with the System Builder.

identifier. This argument is a variable of type AXV$ that (if this call is successful) receives a longword identifier to be used to identify this device in

subsequent calls to AXV_INITIALIZE, AXV_READ, and AXV_WRITE.

maximum_values. This argument is an INTEGER expression that supplies the maximum number of data values that can be read from this device in a single call to AXV_READ.

clock_start_enable. This optional argument is a BOOLEAN expression. TRUE enables a conversion to be initiated by the KWV clock option. The default value is FALSE.

external_start_enable. This optional argument is a BOOLEAN expression. TRUE enables a conversion to be initiated by an external signal. The default value is FALSE. If both clock_start_enable and external_start_enable are FALSE, conversions will be enabled under program control, by a call to AXV_READ.

re_initialize. This optional argument is a BOOLEAN expression. TRUE means that the device has been initialized previously, in which case device_name is ignored and identifier is used to identify the device. No new data structures or objects are created unless the maximum_values argument is greater than the previous value for this device. The default value is FALSE.

use_polling. This optional argument is a BOOLEAN expression. TRUE causes the device to be driven by polling rather than interrupts; FALSE means interrupts will be used to gather data. Polling is always done at device IPL (4 for this device). The default value is FALSE. Polling is only recommended if clock or external starting is being used to initiate conversions; it should not be used if conversions are being initiated under program control.

status. This optional argument is an INTEGER variable that receives the completion status of AXV_INITIALIZE. The only possible value is **1**, which indicates that the procedure completed successfully.

AXV_READ

The AXV_READ procedure causes analog data to be sampled from specified channels, converted to binary form by the device, and stored in a data array. The procedure performs one read for each desired channel, continuing until all data has been collected. It can be called only from programs running in kernel mode. To use the procedure, you must include the module $AXV_UTILITY in the compilation.

Call Format

```
ELN$AXV_READ(
    identifier,
    start_channel,
    end_channel,
    reads_per_channel,
    data_array_ptr,
    kwv_ident,
    gain_array,
    status
    )
```

Arguments

identifier. This argument supplies an expression of type AXV$ that identifies the device to be read; this value is the one returned in the identifier parameter after a call to AXV_INITIALIZE.

start_channel. This argument is an INTEGER expression that supplies the first analog channel to be read.

end_channel. This argument is an INTEGER expression that supplies the last analog channel to be read.

reads_per_channel. This argument is an INTEGER expression that supplies the number of data to be gathered from each channel.

data_array_ptr. This argument is a variable of type ↑ AXV$DATA_ARRAY that receives the address of an array containing converted data from the device. (The meaning of the converted data depends on the positions of several hardware jumpers, as described in the *LSI-11 Analog System User's Guide.*) The first array element corresponds to the first channel read. All or part of this array may be overwritten by subsequent calls to AXV_READ for this device.

kwv_ident. This optional argument is a value of type KWV$ that supplies the identifier of a KWV real-time clock device; this value is the one returned in the identifier parameter after a call to KWV_INITIALIZE. If kwv_ident is present, it is assumed that the KWV device's clock overflow is connected to the AXV/ADV's clock start line. Just before the data is sampled, the clock is started, and it is stopped when all data has been gathered. The KWV device must have been initialized to operate in mode 1 (if more than one value is to be read) or mode 0 (if only one value is to be read) and set up with the desired tick count (which controls how often an overflow is generated) by a call to KWV_WRITE. The call to KWV_WRITE must also have specified st2_go_enable as TRUE so that the call to AXV_READ will do the actual starting of the clock. If kwv_ident is not present, the call to AXV_READ does nothing to start a real-time clock.

gain_array. This optional argument is an array variable of type AXV$GAIN_ARRAY that supplies the

gain to be used in the data conversion for each channel being read. The first array element corresponds to the first channel to be read. The allowable values for this argument are 1, 2, 4, and 8, which are specified by the enumerated type AXV$GAIN. If this argument is not present, the gain value that was used for the last conversion from this AXV device will be used. If no gains were ever used on this device, its initial hardware value of 1 will be used.

status. This optional argument is an INTEGER variable that receives the completion status of AXV_READ.

Status Values

1. This value indicates that AXV_READ completed successfully.

ELN$_AXV_DEVICE_ERROR. This value indicates that either the sampling rate is too high and the data is subject to error, or a conversion was finished before the previous conversion's data was read. Both of these conditions can occur only if conversions are being initiated by the clock or an external signal.

AXV_WRITE

The AXV_WRITE procedure causes a value to be written to an analog-to-digital conversion output register on an AXV11C device. These registers are not present on an ADV11C device; therefore, this procedure cannot be called for an ADV11C device. The result of calling this procedure is to cause an analog output voltage to be generated on the specified channel. It can be called only from programs running in kernel mode. To use the procedure, you must include the module $AXV_UTILITY in the compilation.

Call Format

```
ELN$AXV_WRITE(
    identifier,
    dac_channel,
    value,
    status
    )
```

Arguments

identifier. This argument supplies an expression of type AXV$ that identifies the device to be written to; this value is the one returned in the identifier parameter after a call to AXV_INITIALIZE.

dac_channel. This argument is an expression of type AXV$DAC_CHANNEL that supplies the output channel to be written to.

value. This argument is an expression of type AXV$DATA that supplies the actual data to be written. The manner in which this value determines the output voltage is determined by some hardware jumper settings, as described in the *LSI-11 Analog System User's Guide.*

status. This optional argument is an INTEGER variable that receives the completion status of AXV_WRITE. The only possible value is **1**, which indicates that the procedure completed successfully.

KWV Device Driver Utility Procedures

The procedures described in this section are used in programs that access KWV11C real-time devices. (The *LSI-11 Analog System User's Guide* contains information on the hardware.) Table 14-6 summarizes these procedures.

Table 14-6. KWV Device Driver Utility Procedures

Procedure	Purpose
KWV_INITIALIZE	readies a KWV device for input and creates all needed data structures.
KWV_READ	reads time values from a KWV device and stores them in a data array.
KWV_WRITE	sets up the KWV11C device to generate the clock-overflow signal.

Notes: To use these procedures, include the module $KWV_UTILITY from the RTLOBJECT library in the compilation of your program.

The following Pascal types are declared in module $KWV_UTILITY for use with the KWV device driver utility procedures:

```
TYPE
   { Modes of operation for the device. }
   kwv$mode = (kwv$mode_zero,
               kwv$mode_one,
               kwv$mode_two,
               kwv$mode_three);
```

```
{ Clock rate settings. }
kwv$clock_rate = (kwv$rate_stop,
                  kwv$rate_1mhz,
                  kwv$rate_100khz,
                  kwv$rate_10khz,
                  kwv$rate_1khz,
                  kwv$rate_100hz,
                  kwv$rate_st1,
                  kwv$rate_line);

{ Clock counter and also buffer/preset register. }
kwv$counter = -32768..32767;

{ Identifiers used externally - one per device. }
kwv$ = ↑anytype;

{ Data packets passed to kwv$read. }
kwv$data_array(value_count : integer) = packed
    array[1..value_count] of [word]kwv$counter;
```

KWV_INITIALIZE

The KWV_INITIALIZE procedure readies a KWV device for input and creates all needed data structures. It can be called only from programs running in kernel mode. This procedure must be called at least once for each KWV device used. (The only reason to call this procedure more than once for a single device is to change the value of a parameter or to stop a device that is running in mode 0 or mode 1.) To use the procedure, you must include the module $KWV_UTILITY in the compilation.

Call Format

```
ELN$KWV_INITIALIZE(
    device_name,
    identifier,
    mode,
    clock_rate,
    maximum_values,
    re_initialize,
    use_polling,
    status
    )
```

Arguments

device_name. This argument supplies a string of up to 30 characters giving the name of the device to be initialized. This name must match the name established with the System Builder.

identifier. This argument is a variable of type KWV$ that (if this call is successful) receives a longword identifier to be used to identify this device in

subsequent calls to KWV_INITIALIZE, KWV_READ, and KWV_WRITE.

mode. This argument is an expression of type KWV$MODE that determines the mode in which the device is to be operated:

- KWV$MODE_ZERO, single interval mode. In this mode, action is initiated by a call to KWV_WRITE. The clock is started by either a Schmitt Trigger #2 signal or the call itself. The clock stops after counting the number of ticks specified in the call to KWV_WRITE. At this time (clock overflow), it interrupts the processor, if that is enabled, and asserts the clock-overflow signal line. Note that interrupting the processor on overflow is not supported by this driver.
- KWV$MODE_ONE, repeated interval mode. This mode is identical to single interval mode, except that when clock overflow is reached, the clock is repeatedly restarted to run for the same interval. Therefore, the device produces repeated signals on the clock-overflow line—and repeated processor interrupts, if that is enabled. Note that interrupting the processor on overflow is not supported by this driver.
- KWV$MODE_TWO, external event or program timing mode. In this mode, used for timing an external event, action is initiated by a call to KWV_READ. The clock's counter is set to zero and is started by either the KWV_READ call or a Schmitt Trigger #2 signal. The clock continues to run, and it's counter value is read each time there is an external signal to Schmitt Trigger #2, until the desired number of values has been read.

  This mode may also be used to time a section of code; that is, the clock may be started and stopped

from program control. In this case, it is started by a call to KWV_WRITE. A subsequent call to KWV_READ stops the clock and reads a single value from its counter, which represents the elapsed time since the write. If a device is used in this way, switches 3 and 4 on dip switch sw3 should be in the "off" position; otherwise, any external signals to Schmitt Trigger #2 may result in incorrect operation.

- KWV$MODE_THREE, external event timing from zero base mode. This mode is identical to mode 2, except that the counter is reset to zero each time its contents are read.

clock_rate. This argument is an expression of type KWV$CLOCK_RATE that supplies the clock frequency to be used. This can be a set crystal-controlled frequency, the A.C. line frequency, or Schmitt Trigger #1 input.

maximum_values. This optional argument is an INTEGER expression that supplies the maximum number of data values that can be read from the specified device in a single call to KWV_READ. This argument is only significant for modes 2 and 3. If this argument is not specified, a default of one is assumed.

re_initialize. This optional argument is a BOOLEAN expression. TRUE means that the device has been initialized previously, in which case device_name is ignored and identifier is used to identify the device. No new data structures or objects are created unless the *maximum_values* argument is greater than the previous value for this device. The default value is FALSE.

use_polling. This optional argument is a BOOLEAN expression. TRUE causes the device to be driven by polling rather than interrupts; FALSE means

interrupts will be used to gather data. Polling is always done at device IPL (4 for this device). The default value is FALSE. This argument is only significant for a device operating in modes 2 or 3 when KWV_READ is called to gather data (that is, not when KWV_READ is called following a call to KWV_WRITE).

status. This optional argument is an INTEGER variable that receives the completion status of KWV_INITIALIZE. The only possible value is **1**, which indicates that the procedure completed successfully.

KWV_READ

The KWV_READ procedure reads time values from a specified device and stores them in a data array. The procedure may only be called for a device that has been initialized to operate in mode 2 or 3. It can be called only from programs running in kernel mode. To use the procedure, you must include the module $KWV_UTILITY in the compilation.

There are two possible cases in which KWV_READ would be called:

- The device is not already running. In this case, the call to KWV_READ either starts the device or sets the device so that a signal from Schmitt Trigger #2 will start it. Subsequently, the specified number of data are gathered (each representing the occurence of a Schmitt Trigger #2 signal), and the clock is stopped.
- The device is already running, having been started by a call to KWV_WRITE. In this case, the call to KWV_READ stops the clock, then reads and returns the clock's counter value.

Call Format

```
ELN$KWV_READ(
    identifier,
    value_count,
    st2_go_enable,
    data_array_ptr,
    status
    )
```

Arguments

identifier. This argument supplies an expression of type KWV$ that identifies the device to be read; this value is the one returned in the identifier parameter after a call to KWV_INITIALIZE.

value_count. This argument is an INTEGER expression that supplies the number of values to be read.

data_array_ptr. This argument is a variable of type ↑KWV$DATA_ARRAY that receives the address of an array containing data from the device. Each output datum is a signed 16-bit integer giving a count of ticks.

st2_go_enable. This argument is a BOOLEAN expression. TRUE causes the clock to begin counting upon receipt of a Schmitt Trigger #2 signal. FALSE causes the call to KWV_READ itself to start the counter. The default value is FALSE. This argument is ignored if the clock is already running.

status. This optional argument is an INTEGER variable that receives the completion status of KWV_READ.

Status Values

1. This value indicates that KWV_READ completed successfully.

ELN$_KWV_DATA_OVERRUN. This value indicates that a Schmitt Trigger #2 event occurred before the driver had finished processing the previous one.

KWV_WRITE

The KWV_WRITE procedure performs differently depending on which mode the device is operating in:

- For devices initialized to operate in mode 0 or mode 1, KWV_WRITE causes the device to generate the clock-overflow signal when the specified number of ticks has occurred. Additionally, if the device was initialized to operate in mode 1, clock-overflow signals will be generated repeatedly after each interval containing the specified number of ticks. The clock can be stopped by calling KWV_INITIALIZE to reinitialize it.
- For devices initialized to operate in mode 2 or mode 3, KWV_WRITE causes the device to begin counting from zero, or wait for an ST2 signal to do so. It is then expected that sometime a call to KWV_READ will be made, which reads the current elapsed time.

The procedure can be called only from programs running in kernel mode. To use the procedure, you must include the module $KWV_UTILITY in the compilation.

Call Format

```
ELN$KWV_WRITE(
    identifier,
    st2_go_enable,
    tick_count,
    status
    )
```

Arguments

identifier. This argument supplies an expression of type KWV$ that identifies the device to be written to; this value is the one returned in the identifier parameter after a call to KWV_INITIALIZE.

st2_go_enable. This argument is a BOOLEAN expression. TRUE causes the clock to begin counting upon receipt of a Schmitt Trigger #2 signal. FALSE causes the call to KWV_WRITE itself to start the counter. The default value is FALSE.

tick_count. This optional argument is a 16-bit, signed, positive integer of type KWV$COUNTER that supplies an interval in clock ticks after which a clock-overflow signal is asserted. This argument has no significance if the device was initialized to operate in mode 0 or 1.

status. This optional argument is an INTEGER variable that receives the completion status of KWV_WRITE. The only possible value is **1**, which indicates that the procedure completed successfully.

DLV Device Driver Utility Procedures

The procedures described in this section are used in programs that access DLVJ1 (formerly DLV11-J) serial line controller devices. Table 14-7 summarizes these procedures.

Table 14-7. DLV Device Driver Utility Procedures

Procedure	Purpose
DLV_INITIALIZE	readies a DLV device line for input and/or output and creates all needed data structures.
DLV_READ_BLOCK	reads characters from a serial line until the specified number of characters is read.
DLV_READ_STRING	reads characters from a serial line until a carriage return character is encountered.
DLV_WRITE_STRING	writes the specified character string to a serial line.

Notes: To use these procedures, include the module $DLV_UTILITY from the RTLOBJECT library in the compilation of your program.

The following Pascal named constant and types are declared in module $DLV_UTILITY for use with the DLV device driver utility procedures:

```
CONST
   { Maximum string length. }
   dlv$max_length = 1024;
```

```
TYPE
    { DLV11 device register formats. }
    dlv$rcsr_register = [word] packed record
        ie:         [pos(6)] boolean;
        done:       [pos(7)] boolean;
        end;

    dlv$rbuf_register = [word] packed record
        dat:        [pos(0)] char;
        parity:     [pos(12)] boolean;
        frame:      [pos(13)] boolean;
        overrun:    [pos(14)] boolean;
        error:      [pos(15)] boolean;
        end;

    dlv$xcsr_register = [word] packed record
        break:      [pos(0)] boolean;
        ie:         [pos(6)] boolean;
        ready:      [pos(7)] boolean;
        end;

    dlv$xbuf_register = [word] packed record
        dat:        [pos(0)] char;
        end;

    dlv$registers = record
        rcsr: dlv$rcsr_register;
        rbuf: dlv$rbuf_register;
        xcsr: dlv$xcsr_register;
        xbuf: dlv$xbuf_register;
        end;
```

```
dlv$region(max_length: integer) = record
    count: integer;
    index: integer;
    max_count: integer;
    string_rcv: boolean;
    string: string(max_length);
    end;

{ Device context record. }
dlv$context_record = record
    dl: ↑dlv$registers;
    rcv_device, xmt_device: device;
    rcv_region, xmt_region:
     ↑dlv$region(dlv$max_length);
    interrupt_rcv: boolean;
    priority: integer;
    end;

{ Device context pointer. }
dlv$ = ↑dlv$context_record;
```

DLV_INITIALIZE

The DLV_INITIALIZE procedure readies a DLV device line for input and output and creates all needed data structures. It can be called only from programs running in kernel mode. This procedure must be called once for each DLV serial line used. Since each line is initialized and handled separately from other lines, each line should have its own device description specified in the target system's System Builder menus. To use the procedure, you must include the module $DLV_UTILITY in the compilation.

Call Format

```
ELN$DLV_INITIALIZE(
    device_name,
    identifier,
    maximum_length,
    string_mode,
    use_polling
    )
```

Arguments

device_name. This argument supplies a string of up to 30 characters giving the name of the device to be initialized. This name must match the name established with the System Builder.

identifier. This argument is a variable of type DLV$ that (if this call is successful) receives an identifier to be used to identify this device in subsequent calls to DLV_READ_STRING, DLV_READ_BLOCK, and DLV_WRITE_STRING.

maximum_length. This optional argument is an INTEGER value that supplies the maximum string or block length, in bytes, that will be read or written. The default value is 256.

string_mode. This optional argument is a BOOLEAN expression. TRUE causes the serial line to be used in string mode; FALSE causes it to be used in block mode. The default value is TRUE. String mode means the input is obtained by calling DLV_READ_STRING and will always be terminated by a carriage-return character, CHR(13). Block mode means the input is obtained by calling DLV_READ_BLOCK and will be fixed-length blocks of data, with no carriage return checking performed.

use_polling. This optional argument is a BOOLEAN expression. TRUE means the read procedures will poll the device register; FALSE means the read procedures will use interrupts. Polling is always done at the device's interrupt priority level, which is 4 for the DLV. The default value is FALSE.

DLV_READ_BLOCK

The DLV_READ_BLOCK procedure causes characters to be read from the serial line until the specified number of characters is read. This procedure should be called to read from the serial line if the *string_mode* argument was FALSE in the call to DLV_INITIALIZE. It can be called only from programs running in kernel mode. To use the procedure, you must include the module $DLV_UTILITY in the compilation.

Call Format

```
ELN$DLV_READ_BLOCK(
    identifier,
    block,
    timeout
    )
```

Arguments

identifier. This argument supplies a value of type DLV$ that identifies the serial line device to be read; this value is the one returned in the identifier parameter after a call to DLV_INITIALIZE.

block. This BYTE_DATA argument receives the characters read from the serial line.

timeout. This optional LARGE_INTEGER argument specifies a time interval that is the maximum time allowed for the block of characters to be read. If the timeout occurs, the block is returned incomplete. The default value is zero, which implies no timeout.

DLV_READ_STRING

The DLV_READ_STRING procedure causes characters to be read from the serial line until a carriage return character is encountered. This procedure should be called to read from the serial line if the *string_mode* argument was TRUE in the call to DLV_INITIALIZE. It can be called only from programs running in kernel mode. To use the procedure, you must include the module $DLV_UTILITY in the compilation.

Call Format

```
ELN$DLV_READ_STRING(
   identifier,
   strng
   )
```

Arguments

identifier. This argument supplies a value of type DLV$ that identifies the serial line device to be read; this value is the one returned in the identifier parameter after a call to DLV_INITIALIZE.

strng. This VARYING_STRING argument receives the character string read from the serial line.

DLV_WRITE_STRING

The DLV_WRITE_STRING procedure causes the specified character string to be written to the serial line. It can be called only from programs running in kernel mode. The characters are not interpreted by this procedure; therefore, any variable-length string can be written. To use the procedure, you must include the module $DLV_UTILITY in the compilation.

Call Format

```
ELN$DLV_WRITE_STRING(
    identifier,
    strng
    )
```

Arguments

identifier. This argument supplies a value of type DLV$ that identifies the serial line device to be written to; this value is the one returned in the identifier parameter after a call to DLV_INITIALIZE.

strng. This STRING argument specifies the character string to be written to the serial line.

DRV Device Driver Utility Procedures

The procedures described in this section are used in programs that access DRV11–J parallel line interface devices. Table 14-8 summarizes these procedures.

Table 14-8. DRV Device Driver Utility Procedures

Procedure	Purpose
DRV_INITIALIZE	readies a DRV device controller for input and/or output and creates all needed data structures.
DRV_READ	reads data words from the specified parallel port.
DRV_WRITE	writes data words to the specified parallel port.

Notes: To use these procedures, include the module $DRV_UTILITY from the RTLOBJECT library in the compilation of your program.

These procedures assume that the user device connected to the DRV11–J asserts the USER REPLY lines when the user device is to inform the DRV11–J that either data is available (for reading by the application program) or that data has been accepted (written by the application program).

The following Pascal named constants and types are declared in module $DRV_UTILITY for use with the DRV device driver utility procedures:

```
CONST
   { Port indices. }
   drv$a = 0;                    { Port A index }
   drv$b = 1;                    { Port B index }
   drv$c = 2;                    { Port C index }
   drv$d = 3;                    { Port D index }


TYPE
   { DRV11 device register definitions. }
   drv$csrlo = [byte] packed record
     case integer of
      0: (cmd:    [pos(0)] 0..255);
      1: (armed: [pos(3)] boolean;
          polled: [pos(4)] boolean;
          rprio:  [pos(5)] boolean;
          input:  [pos(6)] boolean;
          noint:  [pos(7)] boolean;
      2: (bits:  [pos(0)] packed array[0..7] of boolean);
          end;

   drv$csrhi = [byte] packed record
         dir:     [pos(0)] boolean;
         ie:      [pos(1)] boolean;
         rdy:     [pos(7)]boolean;
         end;

   { Data buffer word. }
   drv$word = [word] 0..65535;

   { Set of port numbers. }
   drv$port_set = packed set of drv$a..drv$d;
```

```
{ Port registers. }
drv$port = packed record
        csrlo: drv$csrlo;
        csrhi: drv$csrhi;
        dbr: drv$word;
        end;

{ Ports A–D. }
drv$registers =
        packed array[drv$a..drv$d] of drv$port;

{ I/O buffer. }
drv$buffer(words: integer) =
        packed array[drv$a..drv$d, 1..words] of
        drv$word;

{ Device communication region. }
drv$region(words: integer) = record
        count: array[drv$a..drv$d] of integer;
        index: array[drv$a..drv$d] of integer;
        buffer_size: integer;
        buffer: drv$buffer(words);
        end;

{ Device context record. }
drv$context_record(words: integer) = record
        drv: ↑drv$registers;
        done_device: array[drv$a..drv$d] of device;
        interrupt_io: boolean;
        priority: integer;
        io_region: ↑drv$region(words);
        end;

{ Device context pointer. }
drv$ = ↑drv$context_record(1);
```

DRV_INITIALIZE

The DRV_INITIALIZE procedure readies a DRV device controller for input and output and creates all needed data structures. It can be called only from programs running in kernel mode. This procedure must be called once for each DRV controller used. To use the procedure, you must include the module $DRV_UTILITY in the compilation.

Call Format

```
ELN$DRV_INITIALIZE(
    device_name,
    identifier,
    buffer,
    buffer_size,
    output_ports,
    use_polling
    )
```

Arguments

device_name. This argument supplies a string of up to 30 characters giving the name of the device to be initialized. This name must match the name established with the System Builder.

identifier. This argument is a variable of type DRV$ that (if this call is successful) receives an identifier to be used to identify this device in subsequent calls to DRV_READ and DRV_WRITE.

buffer. This argument is a variable of type ↑DRV$BUFFER(buffer_size) that receives a pointer to the I/O buffer. The buffer is a two-dimensional array of data words; the first array index specifies the port number and the second index specifies a data word

number. The I/O buffer is allocated by the DRV_INITIALIZE procedure; it will receive all data read from the device and should be filled with all data to be written to a port.

buffer_size. This argument is an INTEGER value that specifies the size (the number of 16-bit words) of the I/O buffer for each port allocated in the buffer array. This argument is also an upper bound on the buffer array's second index.

output_ports. This argument specifies the set of port numbers of type DRV$PORT_SET that are to be used for output instead of input. The type DRV$PORT_SET defines the SET OF 0..3 for this argument. If a port is specified as an output port, the port's "DIR" bit is set in the port register; otherwise it is cleared.

use_polling. This argument is a BOOLEAN expression. TRUE means the read procedures will poll the device register; FALSE means the procedures will use interrupts. Polling is always done at the device's interrupt priority level, which is 4 for the DRV11-J.

DRV_READ

The DRV_READ procedure causes data words to be read from the specified parallel port. It can be called only from programs running in kernel mode. The resulting data is stored in the buffer pointed to by the buffer parameter returned by DRV_INITIALIZE. To use the procedure, you must include the module $DRV_UTILITY in the compilation.

Call Format

```
ELN$DRV_READ(
   identifier,
   prt,
   word_count
   )
```

Arguments

identifier. This argument supplies a value of type DRV$ that identifies the device to be read; this value is the one returned in the identifier parameter after a call to DRV_INITIALIZE.

prt. This argument supplies an INTEGER value (0..3) specifying which port to read.

word_count. This argument supplies an INTEGER value specifying the number of 16-bit words to be read.

DRV_WRITE

The DRV_WRITE procedure causes data words to be written to the specified parallel port. It can be called only from programs running in kernel mode. Before calling this procedure, the data words should be stored in the buffer pointed to by the buffer parameter returned by DRV_INITIALIZE. To use the procedure, you must include the module $DRV_UTILITY in the compilation.

Call Format

```
ELN$DRV_WRITE(
   identifier,
   prt,
   word_count
   )
```

Arguments

identifier. This argument supplies a value of type DRV$ that identifies the serial line device to be written to; this value is the one returned in the identifier parameter after a call to DRV_INITIALIZE.

prt. This argument supplies an INTEGER value (0..3) specifying which port will be written to.

word_count. This argument supplies an INTEGER value specifying the number of 16-bit words to be written.

Chapter 15
Input and Output

This chapter discusses VAXELN Pascal files, including textfiles and "internal files" used as data structures, and their use in file I/O, record-oriented device I/O, and circuits. The chapter then describes in detail the VAXELN Pascal I/O routines and the VAXELN file utility, disk utility, and tape utility procedures.

Files

In VAXELN Pascal, you can declare files with components of any type except file types or record and array types that have components of file types (see Chapter 3, "Data Types"). Files have the additional properties of a mode, a buffer variable, and a current position. In addition, files are either "open" or "closed."

Open Files and Closed Files

Initially, all files are closed. For typical usage, files can be "opened," usually to associate them with a file on a disk or other mass storage device, with a circuit port, or with a record-oriented device itself, such as a terminal. When a file is open, the records of the file correspond to the components of the Pascal file type or, in the case of the special file type TEXT, to lines of text. Files can be opened either explictly or implicitly.

When a file is not associated with an external device, it is referred to as an "internal file" and can be used as an in-memory data structure in which components of the same type can be accessed sequentially (in memory). In

this case, the entire structure has no specified length, only a defined beginning and an end-of-file position.

Explicit Opening of Files

You can use the OPEN procedure to open a file explicitly, which establishes an association with either a file, device, or circuit. In the common case, a disk file, OPEN can set up access to an existing file or can create a new one. A variety of options are possible when files are opened explicitly for specifying the record characteristics of files and, in particular, whether the files are usable with the direct access procedures, FIND and LOCATE.

Implicit Opening of Files

A closed file can be opened implicitly by the RESET and REWRITE procedures. The sort of input/output connection made is determined as follows:

1. If the file is named as a program parameter, it is opened with the corresponding program argument (a string) as its file specification.
2. If 1 does not apply, but the file is the default textfile INPUT or OUTPUT, it is opened with a default specification, usually referring to the console terminal.
3. If neither 1 nor 2 applies, the file is "opened" for internal I/O only.

Further, if INPUT is closed and it is accessed by some procedure other than RESET (for example, EOF), it is opened implicitly, by the implicit application of RESET to it. Similarly, OUTPUT can be opened by an implicit application of REWRITE if, for example, it is accessed by the PUT procedure while it is closed.

Closing Files

The CLOSE procedure is provided to close files explicitly; such files can be reopened with OPEN, perhaps to give them different characteristics for further I/O processing.

Any file variables declared in a block that are still open when the block terminates normally are closed implicitly.

If you remove a process with the EXIT or DELETE procedures, implicit file closing occurs according to the following rules:

- Removal of a subprocess with the EXIT or DELETE procedure does not close any files implicitly.
- All the job's files are closed if the master process calls EXIT. However, local files of subprocesses are closed only as described in the next item.
- All the job's files are closed if the master process (and thus, the job) is deleted with the DELETE procedure. However, the closing in this case occurs only because all the job's circuits (some of which are used for file I/O) are disconnected. This means that, in this case, any information buffered by the run-time library routines is not flushed to the appropriate device driver or service and so is lost. (This method of exiting a job is not recommended.) Note that freeing a file variable (for example, by the DISPOSE procedure) when it is open has unpredictable effects, although the file will be closed at job termination as described above.

Mode

An open file's *mode* determines whether the components are being generated or inspected. That is, *inspection mode* specifies that a file's records can be inspected, or "read," but no new records can be added. *Generation mode* specifies that new records can be added but no records can be inspected.

The mode of a file is set by the various procedures that operate on files, such as RESET, REWRITE, and (when the "append" argument is TRUE), OPEN.

Buffer Variable

A *buffer variable* holds the value of (at most) one component at a time. Buffer variable references have the format *file-variable* ↑, similar to identified pointer variables. For the file type FILE OF *component-type*, there is an automatically created buffer variable of type *component-type*. For the type TEXT, the buffer variable type is CHAR.

When you create a file on a disk or other record-organized device, the record size is related to the type of the buffer variable, as explained below.

Textfiles

Textfiles have buffer variables of type CHAR and normally have variable-length records. Each record corresponds to a line of the textfile, as explained later in this section. It is also possible to open a file with fixed-length records and use it as a textfile.

FILE OF *type*

When a new file is created, its record size is the size of *type* in bytes; if *type* is a bit-sized PACKED RECORD or PACKED ARRAY, its size is rounded up to the next byte in computing the record size.

If *type* is VARYING_STRING(*n*), the created file has variable-length records with a maximum record size of *n* bytes. Otherwise, the file has fixed-length records.

With READ and WRITE, the file's record size must match the size of *type* unless *type* is VARYING_STRING. In this case, WRITE writes a record whose length is equal to the current length of *file*↑. READ sets the current length of *file*↑ to the record length.

Current Position

The *current position* can be thought of both as a position with respect to the beginning of the file and as defining the contents of the buffer variable.

The position following the last file component is called *end-of-file*. An inspection mode file is at this position after its last component has been read or when it is empty. Normally, a file that is in generation mode is, by definition, positioned at end-of-file because new components are added at the end. The direct-access procedure LOCATE can move a generation-mode file to any position; the next output is written at this position. The function EOF tests for the end-of-file position.

The structure of a nonempty file is shown in Figure 15-1.

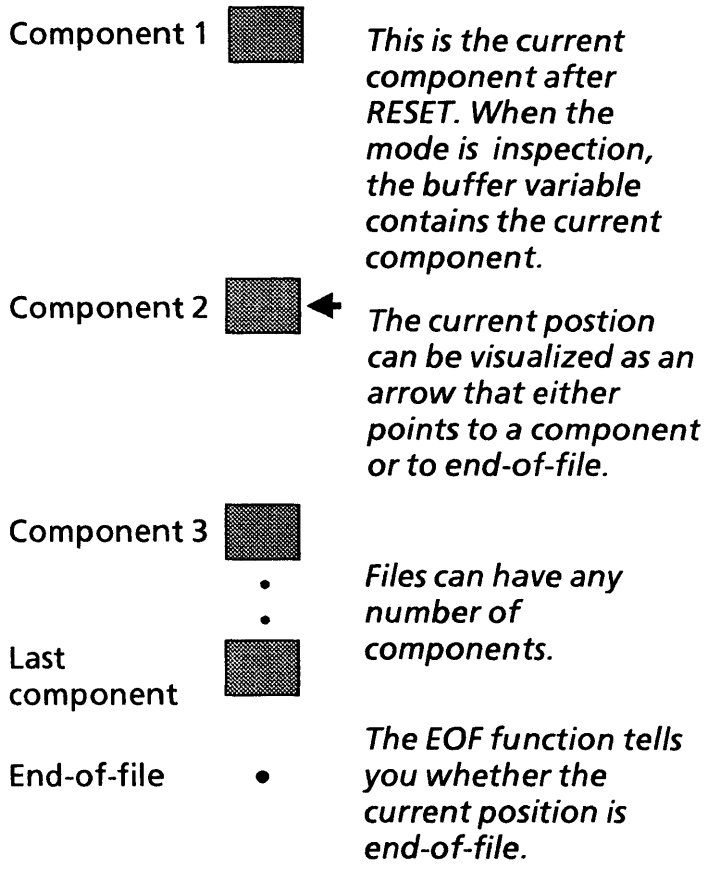


Figure 15-1. Structure of a File

When files are empty, the current position is end-of-file. This is the situation after the REWRITE procedure is performed on a file (which also sets its mode to generation) and after the RESET procedure is performed on an empty file (which sets its mode to inspection).

Inspection Mode and GET

With inspection mode files, the buffer variable normally contains contents of the component at the current position.

Note: With textfile I/O from a terminal, the system does not actually read from the terminal until necessitated by a reference to the buffer variable or the use of EOF, EOLN, and so forth.

You can obtain the current component of an inspection mode file by assigning the buffer variable to a program variable of a compatible type. The GET procedure advances an inspection mode file to its next component, if one exists.

Generation Mode and PUT

With generation mode files, you assign to the buffer variable the value to be added as a new component; then, with the PUT procedure, you append the buffer variable's contents to the file at the current position. After a PUT operation, the buffer variable's contents are undefined.

READ and WRITE

The procedure READ performs the combined action of obtaining an inspection mode file's current component and advancing to the next component. Similarly, WRITE assigns a new value to the buffer variable and writes the buffer variable to a generation mode file in a single procedure call.

Files as Data Structures

As mentioned previously, "internal files" are simply data structures that can be accessed sequentially; their components are allocated the same way as items allocated by NEW. Applying REWRITE or CLOSE to such a file variable disposes of all its associated storage.

TEXT Files

TEXT is a special, predeclared file type with components of type CHAR. Files declared with type TEXT are called *textfiles*. Textfiles have the special properties described below that make them suitable for representing written text and, when they are open, for display on a terminal, printing on a line printer, and so forth. The structure of a nonempty textfile is illustrated in Figure 15-2.

Lines

A textfile is organized as a sequence of *lines* of text (in general, a record is a line). A line is, by definition, a possibly empty sequence of characters terminated with a special component called *end-of-line*.

In textfile I/O, the components of a file are considered to be the individual characters. Thus, GET and PUT transmit single characters. However, this transmission is from or to a hidden buffer maintained by the run-time system; the run-time system transmits whole lines from or to the external file.

In textfile output, the insertion of end-of-line components (that is, the division into records) is determined by the use of the WRITELN procedure. However, attempting a PUT when the hidden buffer is full automatically performs a WRITELN (that is, writes a record) before the PUT.

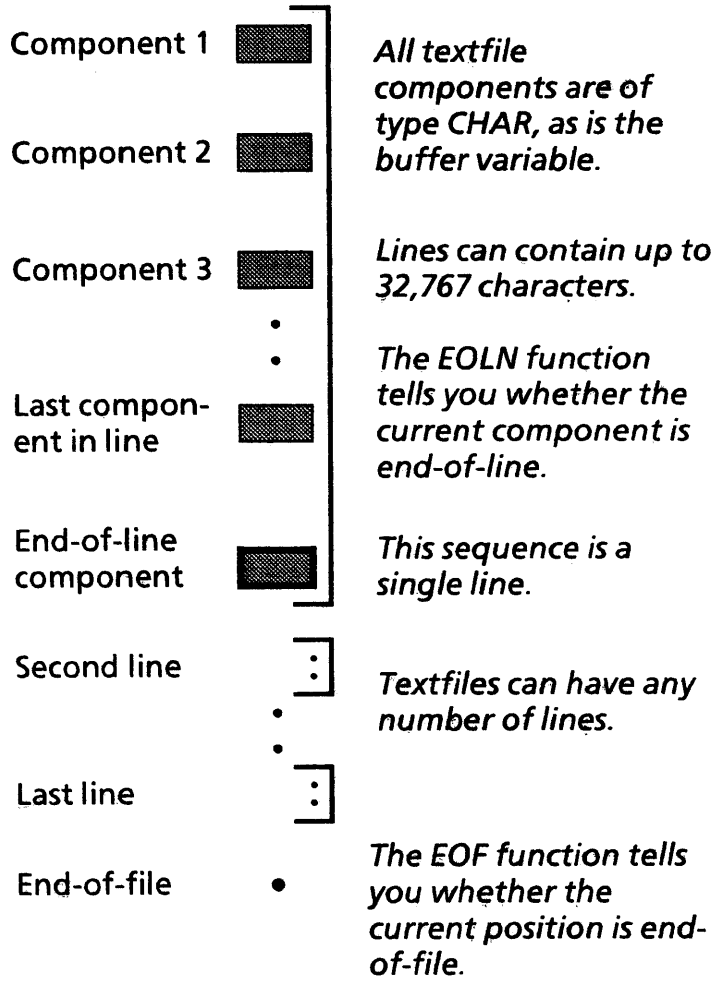


Figure 15-2. Structure of a Textfile

Note that the end-of-line component is generally not a data byte in a file system file. Also note that in inspection mode, if a file is at an end-of-line component, the buffer variable contains a blank.

The predeclared function EOLN can determine whether a textfile's current component is end-of-line. (The end-of-line component cannot be distinguished from a space character except with EOLN and the procedures RESET, READLN, WRITELN, and PAGE.)

The standard files INPUT and OUTPUT are predeclared with type TEXT. (They also are the default files for most file-handling procedures and functions, such as EOF, READ, READLN, WRITE, WRITELN, and EOLN.)

Unless you specify otherwise, these two files also are associated with the console terminal, assuming that your target system has one and was configured with the console driver. That is, the READ/READLN and WRITE/WRITELN procedures by default read and write text messages on this terminal, treating it as a textfile. (Note that the remote debugger will act as a console driver if there is no console driver provided in the system.)

Textfiles in Inspection Mode

When textfiles are in inspection mode, by definition they are either completely empty or else contain at least one complete line. Note that a "complete line" can consist of zero characters (the empty sequence) followed by the end-of-line component.

Textfiles in Generation Mode

When textfiles are in generation mode, the last complete line may be followed by a sequence of

character components that is *not* terminated by end-of-line (that is, a sequence that is an incomplete line). This is simply the state of a generation mode file while a new line of characters is being added.

The various procedures that can change the file's mode to inspection also assure that the last character sequence is properly terminated by end-of-line.

Operations on Files

You operate on files by referring to the buffer variable and by use of the standard routines GET, WRITELN, and so forth. These procedures and functions are categorized below.

The buffer variable associated with a file can be assigned to or referred to regardless of the file's mode. For the sake of avoiding invalid references, note that *the contents of the buffer variable are undefined in the following circumstances*:

- Immediately after a REWRITE operation, since it erases the file's contents.
- Immediately after a RESET operation on an empty file.
- Immediately after any PUT operation, WRITE operation, or WRITELN operation.
- Immediately after a GET, READ, or READLN operation that advances to end-of-file (for READ and READLN, this means that the last record of an open file has been read).

The EOF function can be used to test for end-of-file regardless of the file's mode.

Operations Affecting the Mode

The following operations affect the mode (inspection or generation) of a file:

- The RESET procedure sets the current position to the first component in the file and places the file in inspection mode.
- The REWRITE procedure erases the file's components and places it in generation mode.
- The FIND procedure positions an open, direct-access file at any record and places the current component in the buffer variable. The file can be in either mode initially and is in inspection mode after the operation.
- The LOCATE procedure positions an open, direct-access file at any record. The file can be in either mode initially and is in generation mode after the operation. The next PUT or WRITE writes at that position, overwriting any record already there. (Writing a sequence of records at an arbitrary location requires repeating the LOCATE ... PUT sequence.)
- The OPEN procedure places the opened file in generation mode when the optional append argument is TRUE.

Inspection Mode Operations

The following operations can be performed if and only if the file is in inspection mode:

- Advancing to the next component with the GET procedure (but note that GET(INPUT) implicitly opens INPUT).

- Reading the current component into a variable with the READ procedure.
- Reading the current line of a textfile with the READLN procedure.

Generation Mode Operations

The following operations can be performed if and only if the file is in generation mode:

- Writing out the buffer variable's contents and advancing the current position with the PUT procedure.
- Adding a new component with the WRITE procedure.
- Writing a new line to a textfile with the WRITELN procedure.
- Designating, for textfiles only, that subsequent output is to appear on the next page if the file is actually printed (PAGE procedure).

Pascal I/O Routines

In VAXELN, Pascal I/O is performed by transparent message transmissions between a program and some external entity, such as a terminal, disk file, or other mass storage device. The Pascal I/O run-time routines completely handle all the message protocol details. Simply write programs according to the usual Pascal conventions and include the appropriate resource service or driver in the final system:

- The File Service handles I/O between the program and all file-storage devices. The File Service consists of a disk File Service and a separate tape File Service. To use the File Service, include in the

system the device driver for the device in use. Disk and tape drivers are linked to the appropriate File Service shareable image when they are developed.

- The Console Driver handles I/O with the console terminal on the target machine. The standard textfiles INPUT and OUTPUT are associated with this terminal by default.
- The Terminal Driver handles I/O with additional terminals connected to serial lines.
- The Network Service routes I/O messages and manages universal names in local area networks, where each node is running its own VAXELN system. Each system in such a case is configured with a Network Service. For example, it is possible for a program on one node to read disk files stored on a different node. The Network Service also supports communication with other DECnet nodes.

For details on the File Service, device drivers, and the Network Service, see the *VAXELN User's Guide*.

When the necessary services or drivers are in place, the run-time routines used by your program (READ, WRITE, and so forth) automatically communicate between your program and the device, using circuits and automatically formatted and transmitted messages. Since circuits are used by the underlying I/O operations, the OPEN procedure also allows you to give file properties to a circuit you have created yourself. You can then transmit messages on the circuit with the usual Pascal I/O routines.

The remainder of this chapter describes in detail the VAXELN Pascal I/O routines and the file utility, disk utility, and tape utility procedures. A brief description of each routine is given, followed by the call format, arguments, and function result (if appropriate).

General I/O Procedures

VAXELN Pascal provides predeclared routines to perform general input/output operations on files. Table 15-1 summarizes these procedures.

Table 15-1. General I/O Procedures

Procedure	Purpose
OPEN	explicitly opens a file.
CLOSE	explicitly closes a file.

OPEN

The OPEN procedure opens a file explicitly. In some circumstances, files are opened implicitly (that is, without calling OPEN). If a file is not associated with an external entity by OPEN or by implicit opening (for a file variable named as a program file), it is available only as an in-memory data structure.

Call Format

```
OPEN(
   file,
   FILE_NAME : = specification,
   HISTORY : = file_history,
   RECORD_LENGTH : = length,
   RECORD_LOCKING : = lock,
   ACCESS_METHOD : = access,
   RECORD_TYPE : = record_type,
   CARRIAGE_CONTROL : = control,
   DISPOSITION : = disposition,
   SHARING : = sharing,
   CIRCUIT : = action,
   APPEND : = append,
   BUFFERING : = buffering,
   BUFFERSIZE : = buffer_size,
   CONTIGUOUS : = contiguous,
   EXTENDSIZE : = extend_size,
   FILESIZE : = file_size,
   TRUNCATE : = truncate,
   FILE_ATTRIBUTES : = file_attributes,
   OWNER : = owner_uic,
   PROTECTION: = protection_value,
   STATUS : = stat
   )
```

Arguments

file. This is a file variable representing the file to be opened. INPUT and OUTPUT are permissible if they are closed.

specification. This optional argument supplies a name for the external item to be opened. If it is omitted, the file specification is the specification supplied in the CREATE_JOB call or System Builder argument associated with the specified file variable (which must be a program parameter). If no such information was supplied, and the file variable is either INPUT or OUTPUT, the file-opening applies to the console terminal; if the file name is neither INPUT nor OUTPUT, the file's name is assumed to be the same as the file variable's and its type is DAT.

File specifications have the following format:

```
node::volume:[directory]name.type;version
```

All punctuation separating parts of the specification is required if the parts are present. The type and version can be separated either by a semicolon or a period. The parts of the specification are interpreted as follows:

node This must be a valid node number or the name of a VAXELN node. No node identification is needed to access a VAXELN File Service volume by its volume name. Further, this field need not be specified if the volume being accessed is on the current node. (See **Note 1**, below.)

volume This may be a volume label (such as DISK$VOLUME), an explicit device name (such as DUA1), or null. If null, the volume being referred to is the default

volume on the current node or the default volume on the node specified by the node field. (See **Note 1**, below.)

directory This is the directory containing the disk file; as in VAX/VMS, it can be suffixed with subdirectory names, separated by periods. If this field is not specified, the default is the master file directory on the specified volume ([0,0]).

name This is the name of the file (0–39 characters).

type This is the type of the file (0–39 characters).

version This is the version number of the file (a decimal number between 0 and 65535).

For example,

```
'[test]file1.dat'
```

refers to the most recent version of the file FILE1.DAT in the directory [TEST] on the default volume for the current node. As another example,

```
'trout::file1.dat'
```

refers to the most recent version of the file FILE1.DAT in the master file directory ([0,0]) on the default volume for node TROUT. If that volume's label is TEST, the resultant filename is:

```
'TROUT::DISK$TEST:[0,0]FILE1.DAT'
```

As another example,

```
'disk$test:[data]file1.dat;23'
```

refers to the file FILE1.DAT, version 23, in the directory [DATA] on the volume DISK$TEST, wherever it happens to reside.

As a final example,

```
'dua0:file1.dat'
```

refers to the most recent version of the file FILE1.DAT in the master file directory ([0,0]) on the local volume DUA0.

Note 1: Default volume names are determined to be the first volume specified in the System Builder's *Edit System Characteristers* menu, under the *Disk/volume names* entry. If no volumes were specified, the default volume for the node is the first volume mounted with the MOUNT_VOLUME procedure.

Note 2: In place of a file specification, you can supply the name of a device, as specified in the System Builder's *Edit Device Descriptions* menu. In this case, program I/O (with READ and WRITE, for example), is performed directly between your program and the device driver. For example, if 'TTA2' was established as the name of a terminal, you can open it for communication with

```
OPEN(tty, FILENAME : = 'TTA2:');
```

where tty is a suitably declared file variable.

file_history. This optional argument specifies how to interpret the file-opening depending on whether the file already exists. The possible arguments are values of the predeclared enumerated type FILE_HISTORY:

```
TYPE
  FILE_HISTORY = (
    HISTORY$OLD,
    HISTORY$NEW,
    HISTORY$UNKNOWN,
    HISTORY$READONLY
    );
```

HISTORY$OLD	This opens an existing file. An error occurs if the file does not exist.
HISTORY$NEW	This creates and opens a new file. This is the default.
HISTORY$UNKNOWN	This opens the file if it exists and creates a new one if it does not exist.
HISTORY$READONLY	This opens an existing file for input only. An error occurs if you try to write to the file.

length. This optional argument is an INTEGER expression that specifies the maximum length in bytes (characters) of each record in a textfile or a file of VARYING_STRING. (For files of other types, the size of the records is fixed at the size of the component data type and cannot be overridden.) The value you specify must be in the range 1–32,767. For new files, the default is 133 bytes; for existing files, the default is the length established when the file was created.

By default, textfiles and files of VARYING_STRING have variable-length records, so the file's records can have any number of characters up to the value length. If you create a textfile or FILE OF VARYING_STRING with fixed-length records, this argument specifies the exact length of each record; that is, any output record must have exactly this length.

lock. This optional argument supplies a BOOLEAN value specifying whether record locking is enabled for the file. The default is FALSE. If the file is shareable, you must enable record locking via this parameter. The following rules apply:

- Record locking is possible only on files with fixed-length records. There is no restriction on the record

length unless the file is on a VAX/VMS system, in which case it must be 512 bytes.

- Records are locked against access by the GET, FIND, and PUT procedures only (not READ or WRITE).
- If your program tries to access a locked record, it waits until the record is unlocked. Records are unlocked when the program that locked the record calls RESET, GET, PUT, or some other procedure that changes the current position in the file.

access. This optional argument specifies the method by which individual records are written or retrieved. The possible arguments are values of the predeclared enumerated type FILE_ACCESS:

```
TYPE
  FILE_ACCESS = (
    ACCESS$SEQUENTIAL,
    ACCESS$DIRECT
    );
```

ACCESS$SEQUENTIAL This allows sequential access. Each retrieval obtains the record following the previously obtained record, and each update adds a record after the previously added one.

ACCESS$DIRECT This allows direct access. With this method, the FIND and LOCATE procedures can be used to position the file at any record, and retrieval or update operations take effect there.

ACCESS$SEQUENTIAL is the default method and can be used with either fixed-length or variable-length

records. ACCESS$DIRECT can be used only with fixed-length records and disk devices. It is possible to open an existing file of fixed-length character records for direct access and manipulate it as a textfile.

record_type. This optional argument specifies the kind of records in the file. The possible arguments are values of the predeclared enumerated type FILE_RECORD_TYPE:

```
TYPE
   FILE_RECORD_TYPE = (
     RECORD$FIXED,
     RECORD$VARIABLE
     );
```

RECORD$FIXED This specifies fixed-length records.

RECORD$VARIABLE This specifies variable-length records.

The default for existing files is the record type specified when the file was created. The default for new files is RECORD$VARIABLE for textfiles and files of VARYING_STRING; otherwise, the default is RECORD$FIXED.

control. This optional argument specifies the carriage control applied to output files when they are listed on a terminal or printer. The possible arguments are values of the predeclared enumerated type FILE_CARRIAGE_CONTROL:

```
TYPE
   FILE_CARRIAGE_CONTROL = (
     CARRIAGE$LIST,
     CARRIAGE$FORTRAN,
     CARRIAGE$NONE
     );
```

CARRIAGE$LIST This specifies single spacing between output records.

CARRIAGE$FORTRAN This specifies that the first character of every record is a carriage control character. The interpretations of characters are as follows:

space	New line (single spacing)
0	New line (double spacing)
1	New page
+	Carriage return (for over-printing)
$	Do not terminate partial line (for prompting)

CARRIAGE$NONE Specifies that the records contain no carriage control information.

The default is CARRIAGE$LIST for all textfiles, including the predeclared textfile OUTPUT, and for files of VARYING_STRING components. For all other files, the default is CARRIAGE$NONE.

disposition. This optional argument specifies the action to take when a disk file is closed. The possible arguments are values of the predeclared enumerated type FILE_DISPOSITION:

```
TYPE
  FILE_DISPOSITION = (
    DISPOSITION$SAVE,
    DISPOSITION$DELETE
    );
```

DISPOSITION$SAVE This specifies that the disk file is saved when closed.

DISPOSITION$DELETE This specifies that the disk file is deleted when closed.

The default is DISPOSITION$SAVE. Note that READONLY files cannot be deleted.

sharing. This optional argument specifies whether the file can be shared with other jobs. The possible arguments are values of the predeclared enumerated type FILE_SHARING:

```
TYPE
  FILE_SHARING = (
    SHARE$NONE,
    SHARE$READONLY,
    SHARE$READWRITE
    );
```

SHARE$NONE This specifies that no other job has access to the file while it is open.

SHARE$READONLY This specifies that other jobs can read the file while it is open but cannot write to it.

SHARE$READWRITE This specifies that other jobs can read and write to the file while it is open.

SHARE$READONLY is the default for files that also have HISTORY$READONLY. SHARE$NONE is the default for files with HISTORY$NEW, HISTORY$OLD, and HISTORY$UNKNOWN.

action. This optional argument indicates that the file variable is being used to represent a circuit instead of a data file. (In this case, the arguments length, access, control, record_type, and sharing cannot be specified; history is ignored.) After the OPEN call, GET and READ operations receive messages from the circuit,

and PUT and WRITE operations send messages on the circuit. The possible arguments are values of the predeclared enumerated type OPEN_CIRCUIT:

```
TYPE
  OPEN_CIRCUIT = (
    CIRCUIT$CONNECT,
    CIRCUIT$ACCEPT
    );
```

CIRCUIT$CONNECT This specifies that the OPEN procedure is to create an unnamed port and connect it to an existing named port, the name of which is supplied by the specification argument.

CIRCUIT$ACCEPT This specifies that the OPEN procedure is to create a named port, where the name is supplied by the specification argument, and wait for a circuit connection request on that port. When the OPEN procedure completes, a circuit is established with the named port as your half.

append. This optional argument is a BOOLEAN expression that specifies whether the file is initially positioned at end-of-file. That is, TRUE means that the file is opened in generation mode; in this case, RESET and REWRITE must not be applied to the file before writing records. FALSE is the default and does not imply the mode of the file, so RESET or REWRITE must be used to establish the mode.

buffering. This optional argument is a BOOLEAN expression that specifies whether file I/O is buffered. The default is TRUE. If the value is FALSE, the output written by a PUT operation is written immediately to

the device (unless the device driver also buffers its input), at the loss of the better performance implicit in buffered I/O.

buffer_size. This optional argument is an INTEGER expression that supplies the size in bytes of the run-time library I/O buffer. The I/O buffer is used to "block" multiple records together in a single I/O message. A large buffer reduces the number of messages that must be exchanged between the run-time library and the File Service or device driver handling the open file. The default size is 4096 bytes; the size should not be less than 512 bytes.

contiguous. This optional argument is a BOOLEAN expression that specifies whether a new file is created contiguously on the output disk volume. FALSE is the default. When a contiguous file is created, the disk space for the file is preallocated on consecutive disk blocks. Therefore, a contiguous file is somewhat more efficient to access than a noncontiguous file. However, a contiguous file cannot be extended after it is created, so the file size must be known when the file is created (the file_size argument must be specified when contiguous is TRUE).

extend_size. This optional argument is an INTEGER expression that specifies the number of 512-byte disk blocks by which the file is automatically extended whenever the current allocation is exceeded. This argument can be specified only for new files. The default extension size is 0, which allows the File Service to determine the default extension quantity.

file_size. This optional argument is an INTEGER expression that specifies the number of disk blocks that the file is initially allocated, when it is created. This argument can be specified only for new files. The

default initial size is 0, which allows the File Service to determine the number of blocks to allocate for the file.

truncate. This optional argument is a BOOLEAN expression that specifies whether the file is truncated to the last-used disk block when it is closed. That is, if the file was allocated a larger number of blocks than needed, by file_size or extend_size, the file is truncated to just the size needed to contain its current records. FALSE is the default.

file_attributes. This optional argument is a variable of type ↑ FILE$ATTRIBUTES_RECORD that supplies a pointer to the file attributes record. (The type is declared in the module $FILE_UTILITY.) If an error occurs on the OPEN, you can determine if the record was allocated by checking the pointer variable for NIL. Note that the record is allocated by OPEN, but must be deallocated by the user. To use this argument, you must include the module $FILE_UTILITY from the RTLOBJECT library in the compilation.

owner_uic. This optional argument is an INTEGER value that specifies the User Identification Code of the owner of the file. If specified as 0, this parameter has no effect. In addition, the parameter only has an effect if the file is being created (that is, HISTORY := HISTORY$NEW or HISTORY$UNKNOWN and the file does not exist).

protection_value. This optional argument supplies a protection code of type FILE$PROTECTION for the file. (The type is declared in the module $FILE_UTILITY.) The protection code is a 16-bit word that is composed of four 4-bit fields. Each field represents a category of users: system, owner, group, and world.

The fields are values of the predeclared enumerated type FILE$PROTECTION_CATEGORIES:

```
TYPE
   FILE$PROTECTION_CATEGORIES = (
     FILE$SYSTEM,
     FILE$OWNER,
     FILE$GROUP,
     FILE$WORLD
     );
```

FILE$SYSTEM This defines the system category protection field.

FILE$OWNER This defines the owner category protection field.

FILE$GROUP This defines the group category protection field.

FILE$WORLD This defines the world category protection field.

Each of the four fields consists of four 1-bit indicators that specify the access denied to the category. These indicators are values of the predeclared enumerated type FILE$PROTECTION_TYPES:

```
TYPE
   FILE$PROTECTION_TYPES = (
     FILE$DENY_READ_ACCESS,
     FILE$DENY_WRITE_ACCESS,
     FILE$DENY_EXECUTE_ACCESS,
     FILE$DENY_DELETE_ACCESS
     );
```

FILE$DENY_READ_ACCESS	If this bit is set in a category's field, users in that category are denied read access to the file.
FILE$DENY_WRITE_ACCESS	If this bit is set in a category's field, users in that category are denied write access to the file.
FILE$DENY_EXECUTE _ACCESS	If this bit is set in a category's field, users in that category are denied execute access to the file.
FILE$DENY_DELETE_ACCESS	If this bit is set in a category's field, users in that category are denied delete access to the file.

Note that this parameter only has an effect if the file is being created (that is, HISTORY := HISTORY$NEW or HISTORY$UNKNOWN and the file does not exist). To use this argument, you must include the module $FILE_UTILITY from the RTLOBJECT library in the compilation.

stat. This optional argument is an INTEGER variable that receives the completion status. An exception is raised if the OPEN procedure does not succeed and this argument is omitted.

CLOSE

The CLOSE procedure closes a file. It is usually used to close a file so it can be reopened with different properties, with the OPEN procedure. Any open files are closed automatically upon normal exit from the block in which they are declared, except files allocated by NEW.

Call Format

```
CLOSE(file)
```

Arguments

file. The argument is a file variable representing the file to be closed.

Input Procedures

The input procedures described in this section apply primarily to files opened for sequential access. However, these procedures can also be used on files opened for direct access. Table 15-2 summarizes these procedures.

Table 15-2. Input Procedures

Procedure	Purpose
GET	advances a file to the next component.
READ	reads successive values from an input file and assigns the values to a list of program variables.
RESET	positions a specified file at its beginning and resets the mode to inspection.

GET

The GET procedure advances a file to the next component. Following the GET operation, the file's buffer variable either contains the next component or is invalid (end-of-file).

With open files, each GET reads a record into the buffer variable or, if the file is a textfile, it reads a single character into the (type CHAR) buffer variable. The size in bytes of a non-textfile's records must equal the size of the buffer variable's type unless the buffer variable is of type VARYING_STRING. In textfile input, if the next component is an end-of-line, a space character is stored in the buffer.

Call Format

```
GET(file)
```

Arguments

file. The argument is a file variable; the file must be a nonempty file that is in inspection mode ("open for input"). The current position, prior to the call, must be such that the GET operation would not advance the file beyond end-of-file. In particular, EOF(file) must be FALSE prior to the GET call or an exception will occur.

READ

The READ procedure retrieves as many records as needed from an open file, beginning with the current record, and assigns suitable values to a list of program variables. The current record is either the one located by the immediately preceding FIND procedure; the one following the record last obtained by GET, READ, or

READLN; or, following RESET, the first record in the file. The procedure works similarly on internal (in-memory) files, reading one or more data items from memory until the list of target variables is satisfied.

Call Format

```
READ(
   file,
   target-list
   )
```

Arguments

file. This optional argument is a file variable; if it is omitted, the operation applies to the standard textfile INPUT. If in this case INPUT is currently closed, it is implicitly opened by READ, and RESET is implied as well, meaning that the current record is the first record in the file. With terminal input, this means simply that the first characters read are the first ones that have not been obtained by a previous READ or other input operation.

target-list. This list supplies one or more target variables, the data types of which are assignment compatible with the type of the file's buffer variable. The variables are separated by commas, and there must not be more in the list than there are input items remaining in the file. The READ procedure assigns a data item, beginning with the current contents of the buffer variable (file↑), to each target variable in the list. The assignments continue until the list of targets is exhausted.

Note: The targets are not considered VAR parameters; therefore assignment compatibility with the buffer variable's type is the only requirement.

Algorithms

The exact meaning of the READ procedure differs depending on the type of the specified file's buffer variable and the types of the target variables. Briefly, the differences pertain to special conversions applied when reading data of types other than CHAR from a textfile, including the default textfile INPUT.

In all cases, READ obtains a suitable value from the specified file, assigns it to the target variable, and advances the file to the next data item. Enough records or (for textfiles) characters are read to obtain values for all the targets in the list; the details are given below.

Because READ implies the GET operation, the preconditions for the GET procedure must hold, namely:

- The file must be in inspection mode ("open for input").
- The file must not be positioned at end-of-file.
- For open files that are not textfiles, the record size in bytes must equal the size of the buffer variable's type unless the variable's type is VARYING_STRING.

In addition, the buffer variable (file↑) must be defined. All these preconditions are in effect after RESET has been performed (either explictly or implicitly) on a nonempty file; for example, it is always valid to read from INPUT when it is associated with the system console or other terminal.

Finally, in all cases, the operation

READ(f,t1,t2,...t_n)

is equivalent to:

```
begin READ(f,t1); READ(f,t2); ... READ(f,tn) end
```

(This is merely a formal description of the variable-length list of targets.)

Textfiles with Integer, Floating-Point, BOOLEAN, and Enumerated Targets

In these cases, READ obtains a representative series of characters from the textfile and assigns it, after the appropriate type conversion, to the target. As many records as needed are retrieved in the search for characters. The appropriate series are as follows:

INTEGER targets. For integer targets, the series of characters is an optionally signed sequence of decimal digits.

REAL and DOUBLE targets. For floating-point targets, the series must conform to the syntax for floating-point constants.

Enumerated-type targets (including BOOLEAN). For enumerated variables, the series must spell the name of a previously declared enumerated type value; uppercase and lowercase letters are considered equivalent. The value can be abbreviated if the abbreviation is unique in the type of the target variable. Because identifiers are limited to 31 characters, a run-time error occurs if a 32nd character is encountered before the input field is properly terminated.

Input "fields." Leading spaces, tabs, and line (record) endings are skipped in the search for the series; the series is considered to end when a character is encountered that is not a valid character in the series. This means, in practical terms, that the textfile input is made up of "fields" of characters, and the fields are

delimited by spaces, line endings, commas, or other characters that do not occur in constants of the target types. For example, an enumerated-type field is considered terminated by the occurrence of any character that is not valid in an identifier (such as a space or tab).

Having obtained the series of characters, READ converts it to the target's data type, assigns the converted value to the target, and advances to follow the field just read.

Textfiles with PACKED ARRAY[1..*n*] OF CHAR Targets

You can also use the READ procedure to read a sequence of characters from a textfile into a packed array of characters. The array must have a lower bound of 1. Successive characters from the file are assigned to elements of the array, in order, until each element has been assigned a value. If any characters remain on the line after the array is full, the next READ operation begins with the next character on the line. If end-of-line is encountered before the array is full, the remaining array elements are filled with spaces. Essentially, this operation is treating the packed array similarly to a variable of type STRING(*n*).

Textfiles with STRING(*n*) and VARYING_STRING(*n*) Targets

With a target of either of these types, *n* characters are read from the textfile into the string variable unless end-of-line is encountered first. For open textfiles, end-of-line is the same as the end of the record or the end of a line of input from a terminal. If end-of-line is encountered first, STRING variables are padded with spaces; VARYING_STRING variables assume a new length equal to the number of characters actually read.

Textfiles with CHAR Targets

One character is read from the textfile for each CHAR variable in the list. Line or record boundaries are crossed as needed to satisfy the list of targets and are read as spaces.

All Other Files (FILE OF INTEGER, *Etc.*)

In these cases, the file's data must have the same data type as the targets, and no conversions are needed or performed by READ. That is, each record contains a single value of the target variable's type, and a record is read and its contents assigned to the variable. As many records are read as there are variables in the target list.

Note: The type FILE OF CHAR is not the same as type TEXT; that is, it is not a textfile. A FILE OF CHAR has one character per record, and one record is read for each variable in the target list, which must, again, have type CHAR.

The operation

```
READ(file,target)
```

is equivalent to:

```
begin target : = file↑ ; get(file) end
```

That is, "assign the current value (contents of file↑) to the target and to the value." As stated previously, the GET operation is invalid if, for example, the file is previously at end-of-file, and so, therefore, is READ.

Following RESET on an open, nonempty file, READ assigns the first record's contents to the first variable, reads the next record, assigns it to the next variable, and so on until either all the target variables have been assigned values or end-of-file is encountered.

RESET

The RESET procedure positions the specified file at its beginning; that is, the first record in a nonempty, open file becomes the current record, and the record's contents become the current contents of the file's buffer variable. The RESET procedure changes a file's mode to inspection. It is commonly used to read data from a file after a previous sequence of operations has written data to it.

Call Format

```
RESET(file)
```

Arguments

file. The argument is a file variable representing the file to be reset. Following the call, the file is in inspection mode ("open for input"). Following the call, the content of the buffer variable (file↑) either is undefined (because the file is empty) or is the first component in the file.

If it is currently closed, the file may be opened implicitly by RESET; the particular sort of I/O connection made is determined as follows:

1. If the file is named as a program parameter, RESET opens it implicitly, with the corresponding argument (a string) as its file specification. If there is no corresponding argument, the file is opened with the parameter's name as its name and file type DAT.
2. If 1 does not apply, but the file is the default textfile INPUT (that is, not a declared file with this name but some other component type), it is

opened with a default specification identifying the console terminal.

3. Otherwise, the file is available only as an internal data structure (that is, it is not used for device I/O). The operation of RESET is the same, however: The file's first data item, if any, becomes the current value of file↑, and so forth.

If the default textfile INPUT is referenced by GET, READ, or READLN, it is opened by the implicit application of RESET, as above.

If the file is a nonempty textfile and does not end in an end-of-line indicator, RESET puts one there before moving to the beginning; this assures that every line in a textfile is properly terminated.

Output Procedures

The output procedures described in this section apply primarily to files opened for sequential access. However, these procedures can also be used on files opened for direct access. Table 15-3 summarizes these procedures.

Table 15-3. Output Procedures

Procedure	Purpose
PUT	appends the contents of a file buffer variable to a file.
REWRITE	erases records in a specified file, positions the file at its end, and changes the mode to generation.
WRITE	adds one or more expressions to a specified file.

PUT

The PUT procedure writes the current contents of a file buffer variable to the associated file, device, or circuit. For file output, it appends a new component to a file at the current position. If the current position is end-of-file, the end-of-file now follows the added component. If the file is open and at end-of-file, PUT appends a record (or, for textfiles, a character) to the file; if immediately preceded by LOCATE (which cannot be used with textfiles), PUT replaces the located record.

Call Format

```
PUT(file)
```

Arguments

file. The argument is a file variable representing the output file. Prior to the call, the file must be in generation mode ("open for output").

For example, the following sequence appends a record to the file personnel:

```
VAR personnel: FILE OF RECORD
    name: VARYING_STRING(80);
    hiredate: LARGE_INTEGER;
    hiresalary, currentsalary: REAL;
    END;
.
.
BEGIN
    REWRITE(personnel);
    { Erase file; position at end. }
    WITH personnel^ DO BEGIN
    { Assign to variable. }
      name : = 'Ozgood Franklin';
```

```
        hiredate : = TIME_VALUE(
          '1-OCT-1981');
        hiresalary : = 12000.;
        currentsalary : = 20000.;
        END;
      PUT(personnel);
      { Append record. }
  .
  .
  END
```

REWRITE

The REWRITE procedure erases any records in the specified file and positions the file at its end; that is, EOF becomes TRUE for the file, and it is ready for output. The procedure changes a file's mode to generation.

Call Format

```
REWRITE(file)
```

Arguments

file. The argument is a file variable representing the file to be rewritten. Following the call, the file is in generation mode ("open for output") and the content of file ↑ is undefined because the file is empty.

If it is currently closed, the file may be opened implicitly by REWRITE; the particular sort of I/O connection made is determined as follows:

1. If the file is named as a program parameter, REWRITE opens it implicitly, with the corresponding argument (a string) as its file specification. If there is no corresponding

argument, the file is opened with the parameter's name as its name and file type DAT.

2. If 1 does not apply, but the file is the default textfile OUTPUT (that is, not a declared file with this name but some other component type), it is opened with a default specification identifying the console terminal.
3. Otherwise, the file is available only as an internal data structure (that is, it is not used for device I/O). The operation of REWRITE is the same, however: The file's data is effectively erased, the current content of file ↑ becomes undefined, and EOF is TRUE.

If the default textfile OUTPUT' is referenced by PUT, WRITE, or WRITELN, it is opened by the implicit application of REWRITE, as described above.

WRITE

The WRITE procedure adds one or more expressions to the specified file. If the specified file is a textfile, expressions that are not string values already are converted to suitable character strings before being written out. With open files that are not textfiles, the procedure writes one record for each expression in the source list. With an open textfile, it writes the converted character string one character at a time, crossing record boundaries if necessary.

The written output in an open file begins at the current record, which is either the one located by the immediately previous LOCATE operation; the record following the one last written by WRITE, PUT, or WRITELN; or, following REWRITE, the first record in an otherwise empty file.

Call Format

```
WRITE(
   file,
   source-list
   )
```

Arguments

file. This optional argument names a file variable to which the new data is added. The file must be in generation mode ("open for output"). (For instance, this precondition is satisfied immediately after a REWRITE operation.) If the file is omitted, the WRITE operation applies to the default textfile OUTPUT; if OUTPUT is currently closed, the procedure opens it implicitly by applying REWRITE. Following a WRITE operation, the mode is still generation, and the content of the buffer variable (file ↑) is undefined.

source-list. This list supplies one or more INTEGER, BOOLEAN, floating-point, character-string, or enumerated-type expressions, separated by commas, if the output file is a textfile. If the output file is not a textfile, the expressions must have a data type that is assignment compatible with the type of file ↑. The expressions in the source list are added, one by one, to the output file, beginning at the current position, until the source list is exhausted.

Field-width specifiers for textfile output. When the output file is a textfile, the sources can be suffixed with field-width specifiers; these give the total width in characters of the output field and, for fixed-point representations of real numbers, the number of fractional digits displayed.

The specifiers have the following formats, where *e* is the expression to be written out:

```
e:width
{ Where e is a character, string, BOOLEAN, INTEGER,
REAL, DOUBLE, or enumerated value. }
```

or

```
e:width:fraction
{ Only where e is of type REAL or DOUBLE. }
```

In both cases, width is an INTEGER expression giving the total width of the output field; fraction is an INTEGER expression giving the number of fractional digits shown for a real number displayed in fixed-point notation.

Note that the total field width includes signs, the letter E, and spaces, as applicable to the source data type.

Algorithms

The exact meaning of WRITE depends on whether the output file is a textfile. Briefly, the differences involve the conversions, performed when writing to textfiles, from the source expressions' types to character string representations. For files of other types, no conversions are performed; the types of the source expressions must be assignment compatible with the type of the file's buffer variable.

Files Other than Textfiles

When f is not a textfile, the operation

```
WRITE(f,s1,s2,...sn)
```

is equivalent to

```
begin WRITE(f,s1);WRITE(f,s2); ... WRITE(f,sn) end
```

where s1 to s_n are expressions of a type compatible with f↑. (This is merely a formal description of WRITE's variable number of arguments.)

The operation

```
WRITE(f,s)
```

where s is an expression of a type compatible with f↑, is equivalent to:

```
begin f↑ := s; PUT(f) end
```

Note that, since the conditions for PUT apply, file f is in generation mode both before and after a WRITE operation, and following a WRITE operation, the current content of f↑ is undefined. The PUT operation writes a single record to the file, containing the value of s.

If the file is open, its record size in bytes must equal the size of f↑ unless f↑ is of type VARYING_STRING.

Textfiles

When WRITE applies to a textfile, the source expressions are in general converted to string values before output; sources that are themselves string values are not converted, although their placement in the field can be controlled with field width specifiers.

If no field width specifiers appear in the source list, default widths are chosen appropriately for each data type, the expressions are converted to representative string values, and the strings are appended to the textfile. (At this point, the converted string value is written character-by-character to the file, as for non-textfiles.)

The following paragraphs describe the conversions and default field widths for each source data type.

CHAR. The representation written to the file is a single character preceded by width−1 spaces. The default value of width is 1 (that is, there is no preceding space).

Strings. The representation written to the file is the string value right-justified in the field. If the string is longer than the specified width, it is truncated on the right. The default field width is the actual length of the string in characters.

INTEGER. The representation written to the file is the decimal representation of the integer; it is signed if and only if it is negative. For integers, any specified field width defines a *minimum* field width; integer representations are never truncated. If the integer representation is shorter than the specified or default field width, the representation is right-justified in the field. If the INTEGER value is zero, the single digit 0 is right-justified in the field. The default field width is 10 characters.

BOOLEAN. The representation written to the file is the uppercase string TRUE or FALSE, right-justified in the field. The default value of width is 6. If a field width is specified and is shorter than the string representation, the string is truncated on the right.

Enumerated type. The uppercase version of the enumerated value's identifier is written out. The default field width is the size in characters of the type's longest identifier, plus 1, up to a maximum of 32. The identifier is right-justified in the field.

REAL, DOUBLE. REAL and DOUBLE values can be written in floating-point notation (if no fraction specifier is included) or in fixed-point notation (if fraction is specified).

The floating-point representation is as follows (note that the parts are not separated by spaces in the actual output):

sign int-digit . frac-digits E esign exp

where

sign (–) is included only if the value is negative and is otherwise a single space,

int-digit is the first digit of the value,

. (decimal point) separates the integral and fractional digits,

frac-digits are the specified number of fractional digits,

E is the letter E,

esign (sign of the exponent) is either – or + as appropriate (+ if the floating-point value is zero), and

exp is an exponent (with a leading zero if appropriate). If the output data is type DOUBLE and the G_floating format is used, a three-digit exponent is used; otherwise, a two-digit exponent is used. The exponent for the value zero is 00.

The number of fractional digits (frac-digits) depends on the actual field width. The actual field width is either the specified field width or 8 (9 for G_floating), whichever is larger. The number of fractional digits is the actual field width minus 7 (7 [8 for G_floating] is the total number of characters used by the other components). The floating-point value is adjusted to have one digit to the left of the decimal point and is rounded to this number of fractional digits.

For example, assume that the REAL variable velocity has the value $-313.4789 \times 10^{-17}$. The operation

```
WRITE(f,velocity:10)
```

appends the following field to textfile f (where the angle brackets are shown here for clarity, to delimit the field):

⟨ – 3.135E – 15⟩

If the number of fractional digits is specified in the source, the fixed-point representation is used:

sign int-digits . frac-digits

where

sign (—) is included only if the value is negative and is otherwise a space,

int-digits are the integral digits (0 is shown for the value zero and for fractions), and

frac-digits are the specified number of fractional digits.

The floating-point value is rounded to this number of fractional digits before it is written out. If the specified or default field width is larger than necessary, the fixed-point representation is right-justified in the field. Otherwise, any specified width actually defines a *minimum* field width; fixed-point representations are never truncated. Note that the floating point value 0 is written 0.0 in fixed-point notation. The default field widths are 12 characters for REAL and 20 for DOUBLE.

Direct Access Procedures

The procedures described in this section are generally valid only on files opened for direct access. Table 15-4 summarizes these procedures.

Table 15-4. Direct Access Procedures

Procedure	Purpose
FIND	positions a file at a specified record for input.
LOCATE	positions a file at any record for output.

FIND

The FIND procedure positions a file at a specified record. The file must not have type TEXT, must have fixed-length records, and must have been explicitly opened with ACCESS$DIRECT. The file can be in either mode (inspection or generation) before the FIND call; after the call, its mode is inspection, it is positioned at the indicated record, and the file's buffer variable contains the record's contents. EOF is undefined after any FIND operation.

Call Format

```
FIND(
   file_variable,
   record_number
   )
```

Arguments

file_variable. This argument is a file variable representing the file.

record_number. This argument is a positive INTEGER expression giving the record number relative to the beginning of the file. Record 1 is the file's first record. You can specify a smaller record number than in the last FIND call, allowing you to move either backward or forward in the file. The record number must not be zero or negative.

LOCATE

The LOCATE procedure positions a direct-access file at any record, so that the next PUT (or WRITE) operation can modify that record. LOCATE can specify a record beyond the end-of-file; if so, LOCATE extends the file by adding undefined records between the original end-of-file and the indicated location.

In general, neither GET nor PUT is valid immediately after a LOCATE–PUT sequence; in particular, when the first LOCATE–PUT writes a record other than at the first position in a file, subsequent writing should be done with more LOCATE–PUT sequences.

Call Format

```
LOCATE(
   file,
   record_number
   )
```

Arguments

file. This argument supplies a file variable that was opened with ACCESS$DIRECT. The file can be in either inspection or generation mode before the call; after the call, its mode is generation.

record_number. This argument is a positive INTEGER expression giving the record number relative to the beginning of the file; it must not be zero or negative. Record 1 is the file's first record.

Miscellaneous Routines

The routines described in this section do not fall into any of the previous categories. They are generally used when dealing with sequential access files; however, they can also be used on files opened for direct access. Table 15-5 summarizes these routines.

Table 15-5. Miscellaneous Routines

Routine	Purpose
EOF	indicates whether a specified file is positioned at end-of-file.
FLUSH	flushes the I/O buffers associated with an open file.

EOF

The EOF function indicates whether a specified file is positioned at end-of-file.

Call Format

RESULT : = EOF(*file*)

Arguments and Result

file. The optional argument is a file variable. If it is omitted, the function applies to the standard textfile INPUT.

The result is the BOOLEAN value TRUE if the file is at end-of-file, otherwise FALSE.

FLUSH

The FLUSH procedure flushes the I/O buffers associated with an open file. For a disk file, this causes any records still residing in the I/O system's buffers to be written to the disk.

Note: To flush all buffers, the file must either be open for direct access, or the BUFFERING parameter on the OPEN procedure must be set to FALSE. (That is, the file cannot be in "file transfer" mode.)

Call Format

FLUSH(file)

Arguments

file. The argument is the Pascal file variable associated with the open file.

Textfile Manipulation Routines

The routines described in this section apply only to the handling of textfiles. Table 15-6 summarizes these routines.

Table 15-6. Textfile Manipulation Routines

Routine	Purpose
EOLN	indicates whether a specified textfile is positioned at end-of-line.
GET_CONTROL_KEY	waits for a control key to be pressed at a terminal and stores it in a buffer variable.
PAGE	writes subsequent output to a textfile on the next page.
READLN	reads successive lines from a textfile and assigns the values to a list of variables.
WRITELN	appends a complete line of text to a textfile.

EOLN

The EOLN function indicates whether a specified textfile is positioned at end-of-line.

Call Format

RESULT : = EOLN(*file*)

Arguments and Result

file. The optional argument is a TEXT variable. If it is omitted, the function applies to the standard textfile INPUT. The buffer variable must be valid when the function is called, and EOF(file) must be FALSE.

The function result is the BOOLEAN value TRUE if the current component is the end-of-line component, otherwise FALSE.

GET_CONTROL_KEY

The GET_CONTROL_KEY procedure waits until a control key from a specified set is pressed at a terminal and stores the actual key pressed in a file's buffer variable. It is useful in writing interactive programs.

Calling the procedure specifying a separate file open to a terminal does not affect any read or write operations that may be in progress or issued in the future for the terminal. That is, the procedure is treated by the terminal drivers as a "parallel stream." If a control key is pressed, any GET_CONTROL_KEY requests are completed first if the new key was specified in the requests, and then the key is added to the type-ahead buffer for any current read request.

Call Format

```
GET_CONTROL_KEY(
   file,
   keys
   )
```

Arguments

file. This argument is a TEXT variable that has been opened to a terminal device). The file must be in inspection mode.

keys. This argument supplies a value of type SET OF 0..31, where each element in the set represents one of the 32 ASCII control characters.

A typical use for this procedure is in a command-driven utility that must be halted if it gets into an illegal or otherwise unstoppable state. In this case, a job can create a process that waits until a CTRL/C is pressed, at which time the process deletes the creator or performs some other clean-up activity.

For example:

```
PROCESS_BLOCK stopper(main_process : PROCESS);
CONST ctrl_c = 3;
VAR t : TEXT;
BEGIN
   OPEN (t, FILE_NAME : = 'CONSOLE:');
   RESET(t);
   GET_CONTROL_KEY(t, [ctrl_c]);
   DELETE(main_process);
END;
```

PAGE

The PAGE procedure causes subsequent output to a textfile to be written on the next page, when the file is printed out.

Call Format

PAGE(*file*)

Arguments

file. The optional argument is a variable of type TEXT; the file must be in generation mode and at end-of-file prior to the call. If the argument is omitted, PAGE applies to the default textfile OUTPUT.

If prior to the call, the file is not empty and its last component is not end-of-line, PAGE performs an implicit WRITELN(file), to ensure that all lines in the textfile are properly terminated.

The current component (file↑) is always undefined after a PAGE operation.

READLN

The READLN procedure reads entire lines of input from a textfile (including, if applicable, the rest of the current line) and assigns input values to a list of variables. For textfiles, whether opened implicitly or explicitly, each line corresponds to one record or one line of input from a terminal. The textfile input can be considered to be made up of input fields, which are series of characters representing values, not necessarily values of type CHAR. The definitions of fields and of the type conversions are the same as for the READ procedure.

The procedure assigns a field, after appropriate type conversion, to each target variable in the list, beginning at the current position in the file. Note that the current position may be in the middle of a line immediately after a previous GET or READ; if so, READLN retrieves the rest of that line and as many new ones as it needs to find values for all the variables in the list. Immediately following a READLN, the textfile is positioned either at the beginning of a new line or at end-of-file.

The assignments continue until either all variables have been assigned values or end-of-file is encountered. Line endings and spaces are skipped until the last target variable has been assigned a value, and the file is then advanced to the beginning of the next line.

Call Format

```
READLN(
   file,
   target-list
   )
```

Arguments

file. This optional argument is a TEXT variable; if it is omitted, the operation applies to the standard textfile INPUT. The rules for implicit opening and application of RESET are the same as for the READ procedure. Only textfiles are valid for input with READLN.

target-list. This list optionally supplies one or more target variables, the data types of which are INTEGER, CHAR, string, floating-point, or enumerated types. The variables are separated by commas, and there must not be more in the list than there are fields remaining in the file. If you do not supply a target list, the effect is to simply advance to the next line of input.

Algorithms

The operation

READLN(file, t1, t2, ... t_n)

is equivalent to:

```
begin READ(file, t1, t2, ... tn); READLN(file) end
```

The operation

READLN(file)

is equivalent to:

```
begin WHILE NOT EOLN(file) DO GET(file); GET(file)
end
```

That is, the READ operations obtain the necessary number of fields from the file to satisfy the target list, and then READLN(file) moves the file to the position following the next end-of-line. This position is either end-of-file or the beginning of the next line of input.

WRITELN

The WRITELN procedure appends a complete line of text to a textfile, including the character-string representations of an optional list of expressions. Expressions that are not string values already are converted to suitable character strings before being written out.

If you specify expressions in WRITELN's argument list, each one is converted to a suitable string value, as with WRITE, and written at the current position in the text-file. Note that the current position can be in the middle of a line if the previous output to the file was by PUT or WRITE. If the textfile is open, record boundaries are crossed as necessary while writing out the strings. When all the strings have been written out, the current

line is terminated, so that the next output after WRITELN begins on a new line. If no expressions are specified, WRITELN writes an empty line to the file.

Because textfiles cannot be opened for direct access (and so, LOCATE is invalid), the first expression written by WRITELN is always written at end-of-file.

Call Format

```
WRITELN(
   file,
   source-list
   )
```

Arguments

file. This optional argument supplies a textfile to which the new data is appended. The file must have type TEXT and must be in generation mode ("open for output"). (For instance, this precondition is satisfied immediately after a PUT, WRITE, or REWRITE operation on a textfile.) If the file is omitted, WRITELN applies to the default textfile OUTPUT; if OUTPUT is currently closed, WRITELN opens it implicitly by applying REWRITE. Following a WRITELN operation, the mode is still generation, the last data item in the file is the end-of-line component, and the current content of the buffer variable (file ↑) is undefined.

source-list. This optional list supplies one or more INTEGER, BOOLEAN, floating-point, character-string, or enumerated-type expressions, separated by commas. The expressions in the source list are appended, one by one, to the output file until the source list is exhausted. Before being written out, each source expression is converted to a suitable string value, as with the WRITE procedure applied to textfiles. If no source expressions are supplied, WRITELN merely

appends the end-of-line component; any unfinished line of output is therefore terminated, and subsequent output appears on the next line.

Field-width specifiers. The sources can be suffixed with field-width specifiers; these give the total width in characters of the output field and, for fixed-point representations of real numbers, the number of fractional digits displayed. The specifiers have the following formats, where *e* is the expression to be written out:

```
e:width { Where e is a character, string, BOOLEAN,
INTEGER, REAL, DOUBLE, or enumerated value. }
```

or

```
e:width:fraction
{ Only where e is of type REAL or DOUBLE. }
```

In both cases, width is an INTEGER expression giving the total width of the output field; fraction is an INTEGER expression giving the number of fractional digits shown for a real number displayed in fixed-point notation. Note that the total field width includes signs, the letter E, and spaces, as applicable to the source data type (see the WRITE procedure for details).

Algorithms

The operation

WRITELN(f,s1,s2,...s_i,...s_n)

is equivalent to:

begin WRITE(f,s1,s2,...s_i,...s_n); WRITELN(f) end

The WRITE operations append each converted source expression (s_i) at end-of-file. The expressions must be convertible to character strings, as described above. The operation WRITELN(f) appends the end-of-line component to file f, leaves f in generation mode, and makes the current contents of f ↑ undefined.

File Utility Procedures

The file utility procedures described in this section are performed by the VAXELN File Service for disk and tape volumes. Table 15-7 summarizes these procedures.

Table 15-7. File Utility Procedures

Procedure	Purpose
COPY_FILE	makes an exact duplicate of the specified file.
CREATE_DIRECTORY	creates a directory on the specified disk volume.
DELETE_FILE	deletes a file from a mounted disk volume.
DIRECTORY_CLOSE	closes an existing directory on a mounted disk volume.
DIRECTORY_LIST	obtains the next filename from a mounted disk directory.
DIRECTORY_OPEN	opens an existing directory on a mounted disk volume in preparation for a directory listing.
PROTECT_FILE	changes the protection of a disk file.
RENAME_FILE	renames a disk file.

Notes: To use these procedures, include the module $FILE_UTILITY from the RTLOBJECT library in the compilation of your program.

The CREATE_DIRECTORY, DELETE_FILE, PROTECT_FILE, and RENAME_FILE procedures are invalid for tape volumes.

The following Pascal types (among others) are declared in module $FILE_UTILITY for use with the file utility procedures:

```
TYPE
  { Directory file definition. }
  eln$dir_file = packed record
      dapd: ↑ dap$r_dapd;
      dir_status: integer;
      server: varying_string(255);
      volume: varying_string(255);
      directory: varying_string(255);
      end;

  { Protection code bit definitions. }
  file$protection_types = (
      file$deny_read_access,
      file$deny_write_access,
      file$deny_execute_access,
      file$deny_delete_access);

  { Protection code set definition. }
  file$protection_set = packed set of
    file$deny_read_access..file$deny_delete_access;

  { Protection field definitions. }
  file$protection_categories = (file$system,
      file$owner, file$group, file$world);

  { Record field definition. }
  file$protection =
      packed array[file$system..file$world]
      of file$protection_set;
```

```
{ File attributes record definition. }
file$attributes_record = packed record
    organization:          file$organization;
    record_format:         file$record_format;
    record_attributes:     file$record_attributes;
    bucket_size:                    [byte] 0..255;
    maximum_record_size:            [word] 0..65535;
    block_size:                     [word] 0..65535;
    fixed_control_size:             [word] 0..65535;
    default_extension_quantity:     [word] 0..65535;
    longest_record_length:          [word] 0..65535;
    first_free_byte:                [word] 0..65535;
    maximum_record_number:          integer;
    file_options:          file$file_options;
    device_characteristics:
                           file$device_characteristics;
    allocation_quantity:            integer;
    highest_block:                  integer;
    end_of_file_block:              integer;
    starting_block_number:          integer;
    revision_number:                integer;
    creation_date:                  large_integer;
    revision_date:                  large_integer;
    expiration_date:                large_integer;
    backup_date:                    large_integer;
    owner:                          integer;
    protection:            file$protection;
    resultant_filename: varying_string(255);
    end;
```

(See the source file FILEUTIL.PAS for more information.)

COPY_FILE

The COPY_FILE procedure makes an exact duplicate of the specified file. To use the procedure, you must include the module $FILE_UTILITY in the compilation.

Call Format

```
ELN$COPY_FILE(
    source_file,
    destination_file,
    status,
    source_file_error,
    block_mode,
    count,
    resultant_source_file,
    resultant_destination_file
    )
```

Arguments

source_file. This argument is a string of up to 255 characters giving the file specification of the file to copy. Wildcard characters are not permitted.

destination_file. This argument is a string of up to 255 characters giving the file specification of the copy. Wildcard characters are not permitted.

status. This optional argument is an INTEGER variable that receives the completion status. An exception is raised if the procedure does not succeed and this argument is omitted.

source_file_error. This optional argument is a BOOLEAN expression indicating which file the error occurred on, if the status code is not odd. TRUE

indicates that the error exists in the source file; FALSE indicates that the error exists in the destination file.

block_mode. This optional argument is a BOOLEAN variable that specifies the mode in which the file was copied. TRUE indicates block mode (that is, the file was copied by blocks); FALSE indicates record mode (that is, the file was copied one record at a time). Note that block mode is more efficient for disk files.

count. This optional argument is an INTEGER variable that specifies the number of blocks or records copied, as determined by the *block_mode* argument.

resultant_source_file. This optional argument is a string of up to 255 characters giving the resultant filename of the source file.

resultant_destination_file. This optional argument is a string of up to 255 characters giving the resultant filename of the destination file.

CREATE_DIRECTORY

The CREATE_DIRECTORY procedure creates a directory on the specified File Service disk volume. The directory must be created on a VAXELN disk volume; the procedure cannot create a directory on a remote non-VAXELN system's volume.

This procedure is invalid for tape volumes. To use the procedure, you must include the module $FILE_UTILITY in the compilation.

Call Format

```
ELN$CREATE_DIRECTORY(
    directory_name,
    status,
    owner,
    resultant_directory_name
    )
```

Arguments

directory_name. This argument is a string of up to 255 characters giving the file specification for the directory to be created. Wildcard characters are not permitted. For example, 'DISK$TEST:[DATA]' creates the directory DATA.DIR in the master file directory on the volume DISK$TEST. Note that the procedure creates only the last directory in the specification; any intermediate directories (as in 'DISK$TEST:[intermediate.last]') must already exist, or the error status ELN$_DNF is returned.

status. This optional argument is an INTEGER variable that receives the completion status. An exception is raised if the procedure does not succeed and this argument is omitted.

owner. This optional argument is an INTEGER value that specifies the User Identification Code of the owner of the file.

resultant_directory_name. This optional argument is a string of up to 255 characters giving the resultant filename of the created directory file.

DELETE_FILE

The DELETE_FILE procedure deletes a file from a mounted disk volume. This procedure is invalid for tape volumes. To use the procedure, you must include the module $FILE_UTILITY in the compilation.

Call Format

```
ELN$DELETE_FILE(
    file_name,
    status,
    resultant_file_name
    )
```

Arguments

file_name. This argument is a string of up to 255 characters giving the file specification of the file to be deleted. Wildcard characters are not permitted. If an explicit version number is not specified, or if no semicolon or period is specified, the most recent version of the file is deleted. For example, 'test.dat;23' designates version 23 is to be deleted; 'test.dat;', 'test.dat.', and 'test.dat' all designate the most recent version of the file is to be deleted.

status. This optional argument is an INTEGER variable that receives the completion status of. An exception is raised if the procedure does not succeed and this argument is omitted.

resultant_file_name. This optional argument is a string of up to 255 characters giving the resultant filename of the deleted file.

DIRECTORY_CLOSE

The DIRECTORY_CLOSE procedure closes an existing directory on a mounted disk volume. This procedure is used to terminate a directory listing prematurely without having to call DIRECTORY_LIST until all of the files in the directory have been exhausted. To use the procedure, you must include the module $FILE_UTILITY in the compilation.

Call Format

```
ELN$DIRECTORY_CLOSE(
    dir_file,
    status
    )
```

Arguments

dir_file. This argument is a variable of type ↑ELN$DIR_FILE that supplies a pointer to the directory file. (The type is declared in the module $FILE_UTILITY.)

status. This optional argument is an INTEGER variable that receives the completion status. An exception is raised if the procedure does not succeed and this argument is omitted.

DIRECTORY_LIST

The DIRECTORY_LIST procedure obtains the next filename from a mounted disk directory. To use the procedure, you must include the module $FILE_UTILITY in the compilation.

Call Format

```
ELN$DIRECTORY_LIST(
    dir_file,
    directory_name,
    file_name,
    status,
    file_attributes
    )
```

Arguments

dir_file. This argument is a variable of type ↑ELN$DIR_FILE that supplies a pointer to the directory file. (The type is declared in the module $FILE_UTILITY.)

directory_name. This argument is a variable of type VARYING_STRING(255) that receives the resultant directory specification (when a wildcard directory specification is used in DIRECTORY_OPEN). That is, if more than one directory is traversed by DIRECTORY_LIST, the directory name will change.

file_name. This argument is a variable of type VARYING_STRING(255) that receives the filename. Note that it receives the filename only, not the volume or directory name.

status. This optional argument is an INTEGER variable that receives the completion status. An exception is raised if the procedure does not succeed and this argument is omitted.

file_attributes. This optional argument is a variable of type ↑FILE$ATTRIBUTES_RECORD that supplies a pointer to the file attributes record. (The type is declared in the module $FILE_UTILITY.)

DIRECTORY_OPEN

The DIRECTORY_OPEN procedure opens an existing directory on a mounted disk volume in preparation for a DIRECTORY_LIST operation. To use the procedure, you must include the module $FILE_UTILITY in the compilation.

Call Format

```
ELN$DIRECTORY_OPEN(
    dir_file,
    search_name,
    volume_name,
    directory_name,
    status,
    server_name,
    file_attributes
    )
```

Arguments

dir_file. This argument is a variable of type ↑ELN$DIR_FILE that supplies a pointer to the directory file. (The type is declared in the module $FILE_UTILITY.) Prior to calling the procedure, you must allocate an ELN$DIR_FILE variable, usually with the NEW procedure.

search_name. This argument supplies a string of up to 255 characters giving a specification of an existing directory to search for. The general form of the string is:

```
disk:[directory]filename.type;version
```

The filename, type, and version can use the "wildcard" characters, % and *, as in VAX/VMS file specifications. The % character matches any character in the corresponding position; the * character matches any

character or string in the indicated positions, including null strings.

For example, the string

```
DISK$TEST:[testdata]*A%%C.*;*
```

matches any specification with a filename of at least four characters, the last being C and the fourth-from-last being A, and any file type or version. Wildcards are not allowed in volume names or, for VAXELN volumes, in directory specifications.

If the directory is on a non-VAXELN (for example, VAX/VMS) volume, the asterisk (*), percent sign (%), and ellipsis (...) can be used in the directory specification. The ellipsis following a directory name matches all directories below and including the named directory.

volume_name. This argument is a variable of type VARYING_STRING(255) that receives the volume name if the procedure is successful.

directory_name. This argument is a variable of type VARYING_STRING(255) that receives the directory name if the procedure is successful.

status. This optional argument is an INTEGER variable that receives the completion status. An exception is raised if the procedure does not succeed and this argument is omitted.

server_name. This optional argument is a variable of type VARYING_STRING(64) that receives the resultant node specification or server process port name.

file_attributes. This optional argument is a variable of type ↑ FILE$ATTRIBUTES_RECORD that supplies a pointer to the file attributes record. (The type is declared in the module $FILE_UTILITY.)

PROTECT_FILE

The PROTECT_FILE procedure changes the file ownership UIC and/or the protection code for a specified disk file. This procedure is invalid for tape volumes. To use the procedure, you must include the module $FILE_UTILITY in the compilation.

Call Format

```
ELN$PROTECT_FILE(
    file_name,
    owner,
    protection,
    status,
    resultant_file_name
    )
```

Arguments

file_name. This argument is a string of up to 255 characters giving the file specification. Wildcard characters are not permitted.

owner. This optional argument is an INTEGER value that supplies the ownership UIC of the file. If this argument is not specified, or is specified as zero, the file ownership is not changed.

protection. This optional argument supplies a protection code of type FILE$PROTECTION for the file. (The type is declared in $FILE_UTILITY.) The protection code is a 16-bit word that is composed of four 4-bit fields. Each field represents a category of users: system, owner, group, and world. The protection field definitions are values of the predeclared enumerated type FILE$PROTECTION_CATEGORIES. Each of the four fields consists of four 1-bit indicators that specify

the access denied each category. These protection code bit definitions are values of the predeclared enumerated type FILE$PROTECTION_TYPES. If this argument is not specified, the protection code is not changed.

status. This optional argument is an INTEGER variable that receives the completion status. An exception is raised if the procedure does not succeed and this argument is omitted.

resultant_file_name. This optional argument is a string of up to 255 characters giving the resultant filename of the file.

RENAME_FILE

The RENAME_FILE procedure renames a disk file. This procedure is invalid for tape volumes. To use the procedure, you must include the module $FILE_UTILITY in the compilation.

Call Format

```
ELN$RENAME_FILE(
    old_filename,
    new_filename,
    status,
    resultant_old_filename,
    resultant_new_filename
    )
```

Arguments

old_filename. This argument is a string of up to 255 characters giving the current file specification. Wildcard characters are not permitted. (To rename several related files, use DIRECTORY_LIST to find them and RENAME_FILE to rename each one.)

new_filename. This argument is a string of up to 255 characters giving the new file specification. The new volume name must be the same as the old one; that is, if the old specification includes a volume name, the new one must either supply the same volume name or no volume name. Any parts of the current specification that are not supplied in this argument are obtained from old_filename.

status. This optional argument is an INTEGER variable that receives the completion status. An exception is raised if the procedure does not succeed and this argument is omitted.

resultant_old_filename. This optional argument is a string of up to 255 characters giving the resultant filename of the old file.

resultant_new_filename. This optional argument is a string of up to 255 characters giving the resultant filename of the new file.

Disk Utility Procedures

The disk utility procedures described in this section are performed by the VAXELN disk File Service. Table 15-8 summarizes these procedures.

Table 15-8. Disk Utility Procedures

Procedure	Purpose
DISMOUNT_VOLUME	dismounts a File Service volume on the specified disk drive.
INIT_VOLUME	initializes a File Service disk for use as a file-structured volume.
MOUNT_VOLUME	mounts a File Service disk for use as a file-structured volume.

Note: To use these procedures, include the module $DISK_UTILITY from the RTLOBJECT library in the compilation of your program.

DISMOUNT_VOLUME

The DISMOUNT_VOLUME procedure dismounts a File Service volume on the specified disk drive. The procedure must be called on the same node that has the File Service. To use the procedure, you must include the module $DISK_UTILITY in the compilation.

A dismounted disk can be opened and used for non-file logical block I/O. Note that the user must have RWED privileges to dismount a volume.

Call Format

```
ELN$DISMOUNT_VOLUME(
    device,
    status
    )
```

Arguments

device. This argument supplies a string of up to 30 characters naming the disk drive; for example, 'DQA1' for drive one on disk controller DQA.

status. This optional argument is an INTEGER variable that receives the completion status.

INIT_VOLUME

The INIT_VOLUME procedure initializes a File Service disk for use as a file-structured volume. Disks must be initialized once before they are used. You can initialize any volume on any node running a VAXELN system, but only if the volume is not mounted or open for logical I/O. The procedure must be called on the same node that has the File Service. To use the

procedure, you must include the module $DISK_UTILITY in the compilation.

This procedure is similar in nature to the VAX/VMS command INITIALIZE, as used for disk volumes. (For additional information, consult the VAX/VMS documentation.)

Call Format

```
ELN$INIT_VOLUME(
    device,
    volume,
    default_extension,
    username,
    owner,
    volume_protection,
    file_protection,
    record_protection,
    accessed_directories,
    maximum_files,
    user_directories,
    file_headers,
    windows,
    cluster_size,
    index_position,
    data_check,
    share,
    group,
    system,
    verified,
    bad_list,
    status
    )
```

Arguments

device. This argument supplies a string of up to 30 characters, giving the device specification of the disk drive; for example, 'DQA1' for drive 1 of controller DQA. The node must be specified explicitly for a drive on another node.

volume. This argument supplies a string of up to 12 characters, giving the volume label for the disk. The only valid characters are: A..Z, 0..9, $, and _.

default_extension. This optional argument supplies a value in the range 0–65,535 (type DSK$UWORD) giving the default extension quantity in blocks for all files on the disk volume. The extension quantity is applied when the size of a file is increased beyond its initial allocation by an update. The default is 5 blocks.

username. This optional argument supplies a string of up to 20 characters, giving a user name to be recorded on the volume. If it is omitted, the default is VAXELN.

owner. This optional argument supplies an integer identifying the volume owner (UIC). The default is %x00010001.

volume_protection. This optional argument supplies a value of type DSK$W_PRO (see the example at the end of this section) that specifies the protection for the volume. If it is not specified, users in all categories (system, owner, group, and world) have RWED (read, write, execute, and delete) access. When you specify protection for an entire disk volume, "execute" privilege implies "create" privilege. Note that the group, share, and system arguments can also be used to specify volume protection.

file_protection. This optional argument supplies a value of type DSK$W_PRO that specifies the default

protection code for all files on the volume. If it is omitted, the system and owner have RWED access, the group has RE access, and the world has no access.

record_protection. This optional argument supplies a value of type DSK$W_PRO that supplies the protection code for records. If omitted, the system and owner have RWED access, the group has read access, and the world has no access. (This feature is not currently used.)

accessed_directories. This optional argument supplies a value in the range 0–255 (type DISK$UBYTE) designating the number of directories that can be cached by the File Service by default. The default is 3.

maximum_files. This optional argument is an integer that supplies the maximum number of files that can exist on a disk. The default is calculated by the procedure based on the size of the disk.

user_directories. This optional argument supplies a value in the range 16–16000 (type DSK$UWORD) specifying the number of entries that are preallocated for user directories. The default is 16.

file_headers. This optional argument is an integer that supplies the number of file headers allocated initially for the index file (the file for the volume's file structure). The maximum value is the same as the maximum_files value. The default is 16.

windows. This optional argument is a value of type DSK$UBYTE in the range 7–80 that supplies the number of mapping pointers to be allocated for file windows. When a file is opened, the mapping pointers are used to describe the logical segments of the file for access. The default is 7.

cluster_size. This optional argument supplies a value in the range 1 to 1/100 the size of the volume (type DSK$UWORD) giving the cluster size. The default is 1

for volumes with less than or equal to 50000 blocks and otherwise 3. The cluster size is the minimum allocation unit for the volume.

index_position. This optional argument supplies a value of type DSK$_POSITION that specifies the position of the index file. Possible values are DSK$_BEGINNING, DSK$_MIDDLE, and DSK$_END. The default is DSK$_MIDDLE.

data_check. This optional argument is a value of the type DSK$_DATA_CHECK that enables or disables data checking on read or write operations. Possible values are DSK$_READ (check following all read operations), DSK$_WRITE (check following all write operations), and DSK$_NOCHECK. The default is DSK$_NOCHECK.

share. This optional argument supplies a BOOLEAN value that designates whether the volume is shareable. The default is TRUE, implying that users in all categories have read, write, execute, and delete privileges. If the argument is FALSE, the default protection is no access for group and world, RWED access for system and owner.

group. This optional argument supplies a BOOLEAN value that designates that the disk volume is a group volume. If it is TRUE, the owner UIC defaults to the group number as specified in the owner argument, and the member number defaults to 0. The default is FALSE. If group is TRUE and share is FALSE, the volume protection is RWED for the group, owner, and system. However, if group and share are both TRUE, the volume protection is RWED for all user categories.

system. This optional argument supplies a BOOLEAN value that designates that the volume is a "system volume." In this case, the default protection is RWED access for all users of the system. Only users with

system UICs (group numbers of 0–10 octal, inclusive) can create directories on system volumes. The default is TRUE.

verified. This optional argument supplies a BOOLEAN value that designates whether the volume has information about where bad blocks are located. The default is TRUE. FALSE means that the procedure should ignore information already on the disk about bad blocks. Some disks do not contain any bad block information (for example, RD51s and RD52s). For these types of disks, this argument should be set to FALSE.

bad_list. This argument is a variable of type DSK$_BADLIST that supplies a list of bad blocks. These are areas on the volume that are known to be faulty and are marked by the procedure so that no data will be written on them. The bad block list specifies a range of either logical or physical block numbers. For physical block numbers, pbn_format must be TRUE; for logical block numbers, it must be FALSE. (See the example below.) The argument is required, although a null list can be specified. To specify a null list of bad blocks, allocate a zero-extent array or typecast the variable to an array of zero extents, for example:

```
bad_list : = bad_block_list::dsk$_badlist(0);
```

status. This optional argument is an INTEGER variable that receives the completion status.

Example

The following Pascal types (among others) are defined in module $DISK_UTILITY:

```
TYPE
   dsk$_category = (dsk$_system, dsk$_owner,
      dsk$_group, dsk$_world);
```

```
dsk$_prot_names = (dsk$v_read,dsk$v_write,
    dsk$v_exec, dsk$v_delete);

dsk$w_pro = packed array
    [dsk$_system..dsk$_world] of dsk$b_pro;

dsk$b_pro = packed set of
    dsk$v_read..dsk$v_delete;

dsk$_badblock = packed record
    case integer of
    0: (start_lbn: integer;
        lbn_count: [word]dsk$uword;
        $$fill0: [word]dsk$uword;
        );
    1: (sector: [byte]dsk$ubyte;
        track : [byte]dsk$ubyte;
        cylinder : [word]dsk$uword;
        pbn_count : [word]dsk$uword;
        pbn_format : boolean;
        $$fill1 : packed array [1..15] of boolean;
        );
    end;

dsk$_badlist(badblocks : integer) =
    array [1..badblocks] of dsk$_badblock;
```

In a program, a volume might be initialized as follows:

```
VAR
  bad_block_list : dsk$_badlist(10);
  volpro : dsk$w_pro;
  i : dsk$_category;
BEGIN
  for i : = dsk$_system to dsk$_group do
    volpro[i] : = [];
  volpro[dsk$_world] : =   [dsk$v_read,
                            dsk$v_write,
                            dsk$v_exec,
                            dsk$v_delete];
```

```
bad_block_list[1].start_lbn : = 26899;
bad_block_list[1].lbn_count : = 1;
bad_block_list[1].pbn_format : = false;

INIT_VOLUME(
  'DMA1',
  'TESTVOLUME',
  default_extension : = 10,
  volume_protection : = volpro,
  windows : = 7,
  bad_list : = bad_block_list::dsk$_badlist(1)
  );
```

(See the source file DISKUTIL.PAS for more information.)

MOUNT_VOLUME

The MOUNT_VOLUME procedure mounts a File Service disk for use as a file-structured volume. The procedure requires that the device and its driver (linked to the File Service) be present in the same system from which it is called. The procedure does not return until the disk is completely mounted. To use the procedure, you must include the module $DISK_UTILITY in the compilation.

Call Format

```
ELN$MOUNT_VOLUME(
   device,
   volume,
   status
   )
```

Arguments

device. This argument is a string of up to 30 characters naming the disk drive on which the volume is to be mounted; for example, 'DQA1' for drive 1 on controller DQA.

volume. This optional argument is a variable of type VARYING_STRING(12) that supplies the volume label. If it is omitted, the procedure simply mounts whatever volume is loaded in the indicated drive.

status. This optional argument is an INTEGER variable that receives the completion status.

Tape Utility Procedures

The tape utility procedures described in this section are performed by the VAXELN tape File Service. Table 15-9 summarizes these procedures.

Table 15-9. Tape Utility Procedures

Procedure	Purpose
DISMOUNT_TAPE_VOLUME	dismounts a File Service tape on the specified tape drive.
INIT_TAPE_VOLUME	initializes a File Service tape for use as a file-structured volume.
MOUNT_TAPE_VOLUME	mounts a File Service tape for use as a file-structured volume.

Note: To use these procedures, include the module $TAPE_UTILITY from the RTLOBJECT library in the compilation of your program.

DISMOUNT_TAPE_VOLUME

The DISMOUNT_TAPE_VOLUME procedure dismounts a File Service tape on the specified tape drive. The procedure must be called on the same node that has the tape File Service. To use the procedure, you must include the module $TAPE_UTILITY in the compilation.

Call Format

```
ELN$DISMOUNT_TAPE_VOLUME(
    device,
    unload,
    status
    )
```

Arguments

device. This argument supplies a string of up to 30 characters naming the tape drive; for example, 'MUA0' for drive 0 on tape controller MUA.

unload. This optional argument is a BOOLEAN expression that specifies whether the tape is unloaded. The default is FALSE, implying that the tape is rewound but not unloaded when the volume is dismounted.

status. This optional argument is an INTEGER variable that receives the completion status.

INIT_TAPE_VOLUME

The INIT_TAPE_VOLUME procedure initializes a File Service tape for use as a file-structured volume that conforms to ANSI standard X3.27-1978. Tapes must be initialized before they are used. The procedure requires the device and its driver (and the tape File Service) to be present in the same system from which it is called. The procedure does not return until the tape is initialized. To use the procedure, you must include the module $TAPE_UTILITY in the compilation.

This procedure is similar in nature to the VAX/VMS command INITIALIZE, as used for tape volumes. (For additional information, consult the VAX/VMS documentation.)

Call Format

```
ELN$INIT_TAPE_VOLUME(
    device,
    volume,
    density,
    status
    )
```

Arguments

device. This argument supplies a string of up to 30 characters, giving the device specification of the tape drive; for example, 'MUA0' for drive 0 on tape controller MUA. The node must be specified explicitly for a drive on another node.

volume. This argument supplies a string of up to 6 characters, giving the volume label for the tape.

density. This optional argument is an INTEGER value that supplies the density (in bytes per inch) that the

tape will be initialized to. If the specified density is not supported, the tape will be initialized to the supported density closest to it; the actual density is returned in this argument. The default density is the highest density supported by the specified tape drive.

status. This optional argument is an INTEGER variable that receives the completion status.

MOUNT_TAPE_VOLUME

The MOUNT_TAPE_VOLUME procedure mounts a File Service tape on the specified tape drive for use as a file-structured volume that conforms to ANSI standard X3.27-1978. The procedure requires the device and its driver (and the tape File Service) to be present in the same system from which it is called. The procedure does not return until the tape is completely mounted. To use the procedure, you must include the module $TAPE_UTILITY in the compilation.

Call Format

```
ELN$MOUNT_TAPE_VOLUME(
    device,
    volume,
    block_size,
    status
    )
```

Arguments

device. This argument supplies a string of up to 30 characters, giving the device specification of the tape drive; for example, 'MUA0' for drive 0 on tape controller MUA. The node must be specified explicitly for a drive on another node.

volume. This optional argument is a variable of type VARYING_STRING(6) that supplies the volume label for the tape.

block_size. This optional argument supplies an INTEGER value that determines the number of bytes in each block of a newly created file. The default is 2048.

status. This optional argument is an INTEGER variable that receives the completion status.

Chapter 16
Program Development

Introduction

A Pascal source program or module is compiled using the VAX/VMS command EPASCAL, which invokes the VAXELN Pascal compiler. The compiler then produces an object module. If the source file specified a complete program, the resulting object module can be given directly to the VAX/VMS linker to produce a program image. Otherwise, it can be used as input to later compilations, continuing until you have formed a set of object modules specifying the complete program. The entire set is then given to the linker to prepare the program image.

The LINK command invokes the VAX/VMS linker and produces program images by combining object modules. The program images are then ready to be included in a VAXELN system. The format of the LINK command and the command arguments are explained in detail in the *VAXELN User's Guide.*

The VAX/VMS librarian is used in program development to maintain libraries of object modules for use as input to the VAXELN Pascal compiler or the linker. The use of the librarian is also explained in the *VAXELN User's Guide.*

This chapter explains the format of the EPASCAL command and the command arguments. In addition, module management is discussed, including module dependencies and consistency checking.

EPASCAL Command

The EPASCAL command invokes the VAXELN Pascal compiler and produces a single object module from a single file of VAXELN Pascal source text.

Format

$ EPASCAL *qualifier-list* file-specification-list

Arguments

The command arguments specify options affecting the entire compilation (*qualifier-list*), the source file, and object modules, if needed to compile the source.

File Specifications

The source file is a single file represented by a standard VAX/VMS file specification. If you do not specify a file type, the compiler uses the default file type PAS. The text of the source file, expanded by inclusion of any files specified via the %INCLUDE construction, must satisfy the syntax requirements for a compilation unit, as explained in Chapter 2, "Program Structure."

By default, the resulting object module has file type OBJ and the same filename as the source file.

Additional specifications can be listed, to designate object libraries or single object modules whose exported declarations can be included during compilation. Multiple file specifications can be separated by commas or by plus signs.

Libraries have the default file type OLB and are designated by the LIBRARY qualifier on the file specification. Object-module files (default type OBJ) are designated by the MODULE qualifier on the file

specification. If /LIBRARY is specified for a file, all subsequent files in the list are treated as object libraries unless /LIBRARY or /MODULE is found on a later file specification. Up to eight object libraries can be listed; there is no limit to the number of object-module files.

Qualifiers

The qualifiers described below can be applied to the EPASCAL command or to the source file specification; each is preceded by a slash (/). All qualifiers and options can be abbreviated to the shortest unique form.

Note: Unless a qualifier is explicitly stated to apply to a file, it applies only to the command. In addition, the LIBRARY and MODULE qualifiers apply only to files other than the first (source file).

The default qualifiers in interactive use are:

```
NOCHECK
NOCROSS_REFERENCE
DEBUG = TRACEBACK
EXPORT
NOG_FLOATING
INLINE
NOLIST
NOMACHINE_CODE
NOMAP
OBJECT
OPTIMIZE
SHOW = (SOURCE, HEADER)
VALIDATE = REQUIRED
WARNINGS
```

CHECK = *(option-list)*, NOCHECK. These qualifiers enable or inhibit checking of various kinds, which are specified by the following options:

ALL	Enables all CHECK options.
ASSERT, NOASSERT	Enables or inhibits detection of false assertions. NOASSERT disables both compile-time and run-time assertion checking.
RANGE, NORANGE	Enables or inhibits run-time detection of most range violations.

NOCHECK is the default. If CHECK is specified without an option list, range and assertion checks are performed by default. (Note also that range violations that can be detected by the compiler are detected irrespective of the RANGE option.)

CROSS_REFERENCE, NOCROSS_REFERENCE. These qualifiers enable or disable generation of a cross-reference listing. For each variable in the program, a cross-reference lists the line numbers containing references. The LIST and default MAP qualifiers are implied by CROSS_REFERENCE. The default is NOCROSS_REFERENCE.

DEBUG = *(option-list)*, NODEBUG. These qualifiers include or omit debugging information in the created object module. DEBUG means that the object module has traceback information, source-line information, and a debugger symbol table with the names of all items declared in the source module. NODEBUG means that none of this information is included.

If neither DEBUG nor NODEBUG is specified, the object module contains traceback and source-line information, but no symbol table.

Note: You must also specify the DEBUG qualifier on the LINK command to transfer the object module's symbol table, if any, to the final program image. In other words, if the EPASCAL command has the explicit DEBUG qualifier, so should the LINK command.

The option list allows more control of the debugger information, as follows:

ALL Include traceback, source lines, and all this module's symbols; equivalent to DEBUG with no option list.

EXPORT_ONLY Include traceback, source lines, and only those symbols exported from the module.

IMPORT_TOO Include traceback, source lines, symbols defined in this module, and referenced symbols imported from other modules. This option is useful for debugging programs using standard definition modules (such as $DAP) that are compiled without symbols.

NONE No information is included for the debugger; equivalent to NODEBUG.

SYMBOLS Include traceback, source lines, and all symbols defined in this module; equivalent to DEBUG with no option list.

TRACEBACK Include traceback and source lines only; this has the same effect as when you specify neither DEBUG nor NODEBUG.

Generally speaking, you should request a listing if you intend to debug a program.

EXPORT, NOEXPORT. These qualifiers enable or inhibit the generation of an export symbol table. This table contains the declaration information for exported names for use in other modules. A module compiled with NOEXPORT cannot be used as input to subsequent EPASCAL compilations. In effect, the result is a non-EPASCAL object module. NOEXPORT is intended only for unusual situations. (See also the DEBUG options above, for information about debugging with exported and imported symbols.)

G_FLOATING, NOG_FLOATING. These qualifiers determine whether DOUBLE data are represented in D_floating (default) or G_floating format. If you are building a system for a MicroVAX I, be sure that you generate instructions only for the double-precision data type supported by the hardware on your machine or else include the floating-point instruction emulation with the System Builder.

INCLUDE = (module-list). This qualifier includes the listed modules in the compilation. (They must be present in the libraries specified in the command or be specified input files using /MODULE.) The INCLUDE qualifier provides a way to include modules in a compilation that are not mentioned in the INCLUDE line of the MODULE heading (that is, within the source file).

INLINE, NOINLINE. These qualifiers enable or disable the INLINE attribute on procedures and functions in the compilation unit. The default is INLINE. (Note that these qualifiers do not affect procedures and functions declared in included modules.)

LIBRARY. This qualifier is used on a file specification to designate that it is an object library containing object modules for input to the compilation. If you specify it for an input file, it applies to subsequent files unless LIBRARY or MODULE is specified again.

LIST = *file-specification*, NOLIST. These qualifiers enable or disable generation of a compiled-source listing. The specified file, if any, receives the listing; by default, the listing file has the name of the source file and type LIS. The default is NOLIST for interactive compilations, LIST for batch compilations. The default LIST qualifier is supplied implicitly when you use any qualifier that requires a listing.

MACHINE_CODE, NOMACHINE_CODE. This qualifier generates a listing with machine code following the corresponding VAXELN Pascal statements. MACHINE_CODE implies the default LIST qualifier. (**Note:** The NOOBJECT qualifier inhibits code generation, and therefore the machine code listing, even if MACHINE_CODE is specified.)

MAP = option, NOMAP. These qualifiers enable or disable listing of a storage map. The default LIST qualifier is implied. The following options are supported, with REFERENCED the default option:

LOCAL Include symbols defined in the source module.

REFERENCED Include symbols defined or referenced in the source module.

ALL Include symbols defined in the source module and all included modules.

MODULE. This qualifier is applied to a file specification to designate that it is an object module for input to the compilation. If you specify it for an input file, it applies to subsequent files unless LIBRARY or MODULE is specified again.

OBJECT = *file-specification*, NOOBJECT. These qualifiers write the object module to the specified file, or inhibit code generation and production of an object

module. If both are omitted, the object file has the name of the source file and type OBJ.

OPTIMIZE = *(option-list)*, NOOPTIMIZE. These qualifiers enable or inhibit various compiler optimizations. All optimizations are performed if no options are listed. NOOPTIMIZE inhibits many of the optimizations performed by the compiler, including some (such as short-circuit evaluation of Boolean expressions) which are not reflected in the option list. With the option list, certain optimizations can be selectively inhibited, although it is seldom appropriate to do so. Multiple options must be separated by commas and the list enclosed in parentheses. The following options are supported:

COMMON_SUBEXPRESSIONS, NOCOMMON_SUBEXPRESSIONS	Enables or inhibits removal of common subexpressions from statement sequences.
DISJOINT, NODISJOINT	Enables or inhibits the placement of local variables in multiple registers. (DEBUG = SYMBOLS implies NODISJOINT.)
INVARIANT, NOINVARIANT	Enables or inhibits removal of invariant expressions from loops.
LOCALS_IN_REGISTERS, NOLOCALS_IN_REGISTERS	Enables or inhibits placement of local variables in registers.

PEEPHOLE, NOPEEPHOLE	Enables or inhibits replacement of code patterns with simplified code.
RESULT_INCORPORATION, NORESULT_INCORPORATION	Enables or inhibits replacement of certain operations with three-operand instructions.

SHOW = *(option-list)*. This qualifier specifies a list of items for inclusion in the listing file. Multiple options must be separated by commas and the list enclosed in parentheses. The following options are supported:

HEADER, NOHEADER	Enables or inhibits page headers.
INCLUDE, NOINCLUDE	Enables or inhibits listing of %INCLUDE files (LIST is implied).
MODULES, NOMODULES	Enables or inhibits listing of module names used in this compilation (LIST is implied).
SOURCE, NOSOURCE	Enables or inhibits listing of VAXELN Pascal source code.
STATISTICS, NOSTATISTICS	Enables or inhibits listing of compilation statistics.

The default SHOW options are SOURCE and HEADER.

VALIDATE = option. This qualifier specifies the level of version consistency checking among modules included in the compilation. (See "Module Management," below.)

The following options are supported, with REQUIRED as the default option:

NONE The compiler tries to generate code even if inconsistencies are detected that might generate incorrect code. An informational message is issued if any are detected.

REQUIRED The compiler issues error-level messages if inconsistencies are detected among modules on which this one depends for declarations.

ALL The compiler issues error messages if inconsistencies are detected among any related modules.

WARNINGS, NOWARNINGS. These qualifiers enable or inhibit compiler diagnostic messages with Warning severity. Errors with this severity usually allow compilation to proceed, although, in some cases, the program will fail if executed. WARNINGS is the default.

Module Management

The treatment of modules and exported symbols in VAXELN Pascal is based on the idea that there should be only one declaration of a given object. Thus, there should be only one declaration of a procedure and its parameter list, even if the procedure is used in many routines in many different programs.

The declaration of an exported name is stored (in compiled form) in the object module containing the code associated with the name. This reduces the number of files that need to be managed, and it helps to avoid inconsistencies that can arise from multiple versions of a module, such as when a module is changed during the course of program or system development.

Object modules can be stored in object libraries for use as compiler input or as direct input to the VAX/VMS linker. In addition, VAXELN supplies run-time libraries for linking with Pascal object modules to form a complete VAXELN system. These libraries are listed in the *VAXELN User's Guide.*

Inclusion of Modules in a Compilation

If a compilation unit, *C*, uses a name exported from another module, *M*, that module must be compiled before *C*, and it must be included in the compilation of *C*. *M* will be included in *C*'s compilation if it is specified as an input file in the EPASCAL command using the MODULE file qualifier. Alternatively, and more typically, *M* can be placed in an object module library, *L*, which is then specified as an input file to the EPASCAL command using the LIBRARY file qualifier. For example:

```
$ EPASCAL C + L/LIBRARY
```

Note that specifying the library, *L*, as an input file does not suffice to include *M* in the compilation. *M*'s inclusion must be requested either by an include header in the compilation unit, an INCLUDE option on the EPASCAL command, or indirectly because *M* is needed by another module in the compilation. If *M* is not included and the compilation unit tries to reference a name exported from *M*, an undeclared-name error message will result. There is no automatic search of libraries to find the module exporting a name.

The exact rules for determining the modules included in a compilation are as follows:

1. Modules provided as EPASCAL input files with the MODULE qualifier are included.

2. Modules specified in the include header of a compilation unit and not already included are included by searching the libraries specified in the EPASCAL command line.
3. Modules specified in the EPASCAL command qualifier INCLUDE and not already included are included by searching the specified libraries.
4. Any related modules not already included are included by searching the libraries. A *related* module, *R*, is a module on which some included module, *M*, depends; that is, *R* was included in the compilation of *M* and *M* uses a declaration exported by *R*. Note that the interpretation of *related* is controlled by the VALIDATE command qualifier, discussed in the next subsection.

Inclusion of a module by library search is the typical method in EPASCAL. The libraries listed as input files are searched in the listed order. The module, identified by its name, is taken from the first library in which it occurs. If the module is not found, the compiler issues an error message.

Note that if a module, *A*, occurs in two libraries (that is, each library contains an object module named *A*), the occurrence in the first library overrides that in the second. For example, in developing a new version of a program, you might use a library, *L2*, containing all modules of the original version and a library, *L1*, containing modified versions of some modules. For example:

```
$ EPASCAL newversion + L1/LIBRARY + L2
```

Note that the LIBRARY qualifier need be specified only once, since it applies to all subsequent files unless another LIBRARY or MODULE qualifier is specified.

Module Dependencies and Consistency Checking

Modules are typically modified and recompiled in the course of program development. This can lead to inconsistencies in compiling and linking the complete program. The VAXELN Pascal compiler, therefore, takes some steps to detect such inconsistencies.

For example, suppose that a complete program contains a module *P*, the PROGRAM block, that uses modules *A* and *B*, and that *B* also uses *A*. Suppose, then, the following sequence of events:

1. *A* is compiled.
2. *B* is compiled.
3. *A* is edited and recompiled.
4. *P* is compiled.

When *P* is compiled, the compiler will (with high probability) report an inconsistency: module *B* depends on a different version of module *A* than the one included in the compilation of *P*. This can be corrected by recompiling *B* and then again compiling *P*.

This subsection explains in more detail how the compiler's consistency checking works, and how the VALIDATE command option may be used to control the checking. It should be noted that the compiler's capabilities are intended to supplement other methods of module management, such as the use of a source code management system or the use of systematic builds in which all modules and programs in a system are compiled and linked in the proper order. (The VAXELN developers use the latter method to ensure system consistency.)

When a module, *M*, is compiled, the compiler computes a "checksum" based on the contents of the module's

exported symbol table. The checksum is 32 bits long and is computed using exclusive OR. If two different versions of the same module have a different exported symbol table, it is highly likely (but not certain) that their checksums will differ. This forms the basis of the compiler's module consistency checking. (Note that even a trivial change to a module, such as interchanging the order of two declarations, may change the checksum. In practice, the compiler's notion of consistency is rather severe.)

You can determine the checksum of a VAXELN Pascal module using the ANALYZE/OBJECT command. The third record of a VAXELN Pascal module will be entitled "IGNORED HEADER (subtype 101)". Bytes 4 through 7 of the displayed data contain the checksum.

Consistency checking via checksum comparison is driven by module dependency information stored in each module's exported symbol table. Module *A* is dependent on module *M* if *A* references any symbol exported from *M*. If the only references are in the executable statements of non-in-line routines, *A* is *code-dependent* on *M*. Otherwise, *A* is *declaration-dependent* on *M*.

If *A* is dependent on *M*, *A*'s exported symbol table contains an entry specifying its dependency on *M* and giving the checksum of the version of *M* included in *A*'s compilation. If any later compilation includes both *A* and *M*, the compiler compares *M*'s actual checksum with that specified in *A*'s exported symbol table (for its dependency on *M*) and reports a difference as a module inconsistency.

Note that the EPASCAL command option SHOW=MODULES may be used to obtain a listing of included modules and their dependencies.

If a detected inconsistency is one involving declaration-dependent modules, the compiler issues an error-level message, which prevents code generation for the current compilation unit. It does this because such declaration-related inconsistencies may lead to a fatal compiler error or to the generation of incorrect code.

As noted above, the compiler's consistency checking can prove rather severe in practice. During program development, it is often desirable to compile each module as it is changed, recompiling other, dependent modules only when really necessary. For example, suppose you add a completely new constant, ALPHA, to a module you are working on. Since no other modules can be invalidated by this new declaration, it would be convenient to avoid recompiling modules that depend on the current one for other reasons.

To support this mode of operation, the VALIDATE=NONE qualifier for the EPASCAL command is provided. With this qualifier, each module inconsistency is reported only as a warning-level message. In addition, one informational message is printed if there are inconsistencies that would normally be reported at the error level. (Note that ignoring a non-harmless module inconsistency in this way may lead to a fatal compiler error or to the generation of incorrect code.) Eventually, one should validate consistency by the normal method or systematically compile all modules in the proper order.

The VALIDATE command qualifier has two additional forms. VALIDATE = REQUIRED is the default. This simply means that the compiler includes in the compilation only those modules explicitly requested and those on which included modules are declaration-dependent. Consistency is checked as described above.

The command qualifier VALIDATE = ALL can be used to force additional consistency checking. It has the following effects:

- If a directly or indirectly included module is code-dependent or declaration-dependent on another module, that module is also included in the compilation.
- If any module inconsistency is detected, an error-level summary message is issued (even if the inconsistency is a warning-level one).
- An error-level message is issued if any module contains a SEPARATE procedure or function declaration without the corresponding separate routine body being in one of the included modules (or the source module).

If you structure your program so that no declarations are exported from the module containing the PROGRAM block, then compiling it with VALIDATE = ALL validates the entire program.

Appendix A
Attributes

This appendix lists the attributes allowed in the syntactic category "Attributes" and the context in which the attributes may be used.

- **UNDERFLOW, NOUNDERFLOW.** Any complete routine declaration. (The routine declaration must contain a routine body; it cannot specify a directive, such as EXTERNAL.)
- **INLINE.** A complete procedure or function declaration containing a routine body.
- **EXTERNAL.** An outer-level VAR declaration.
- **VALUE.** A VAR declaration (not parameter).
- **READONLY.** A VAR declaration (not parameter) or a value parameter.
- **REFERENCE.** A value parameter (in a procedure or function parameter list) whose data type is such that the parameter would normally be passed by immediate value.
- **OPTIONAL.** A VAR parameter, or a procedure or function heading occurring as a parameter.
- **LIST.** The last parameter in a parameter list.
- **BIT, BYTE, WORD, LONG.** A PACKED record definition, an ordinal type, or a small set type.
- **ALIGNED.** An array type definition, a record definition, or a field in a record.
- **POS.** A field in a PACKED record.

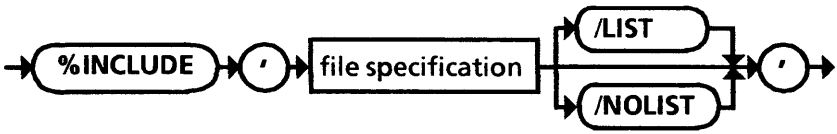
Appendix B
Collected Syntax

This appendix is an alphabetical collection of syntax diagrams representing the syntactic categories of the VAXELN Pascal language. These categories are explained individually throughout this manual, in the appropriate sections.

Chapter 1, "Notation and Lexical Elements," contains an explanation of the conventions used in these syntax diagrams. Table 1-1 lists the VAXELN Pascal reserved words; Table 1-2 defines the complete set of punctuation symbols used as delimiters in VAXELN Pascal programs; Table 1-3 summarizes the special symbols used as operators.

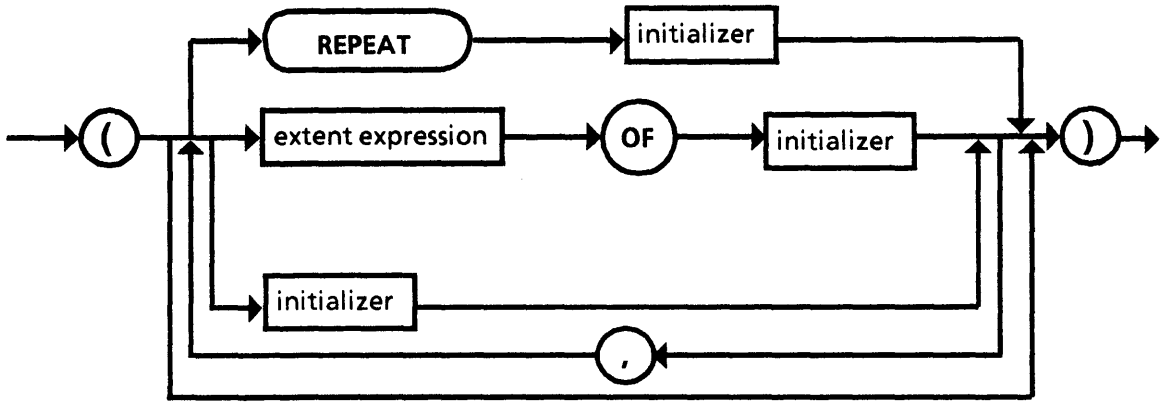
As explained in the syntax conventions, the terminal syntax elements in square boxes are defined by lexical rules. The rules for identifiers are given in Chapter 1 and the rules for literal constants are given in Chapter 4, "Constants."

Note that in order to include text from other files as part of the source, the source file for a compilation may contain constructions of the following form:

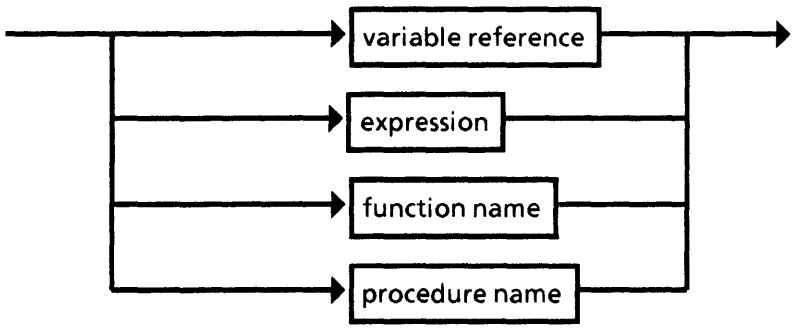


The relation of the syntax to the operation of the compiler is as follows: A complete VAXELN Pascal program consists of one or more compilation units, one of which contains a PROGRAM block declaration. An invocation of the compiler compiles one source file, satisfying the syntax for the compilation unit. One object module is produced. The compilation may reference information in other object modules.

Aggregate Initializer



Argument



(for VAR parameters)

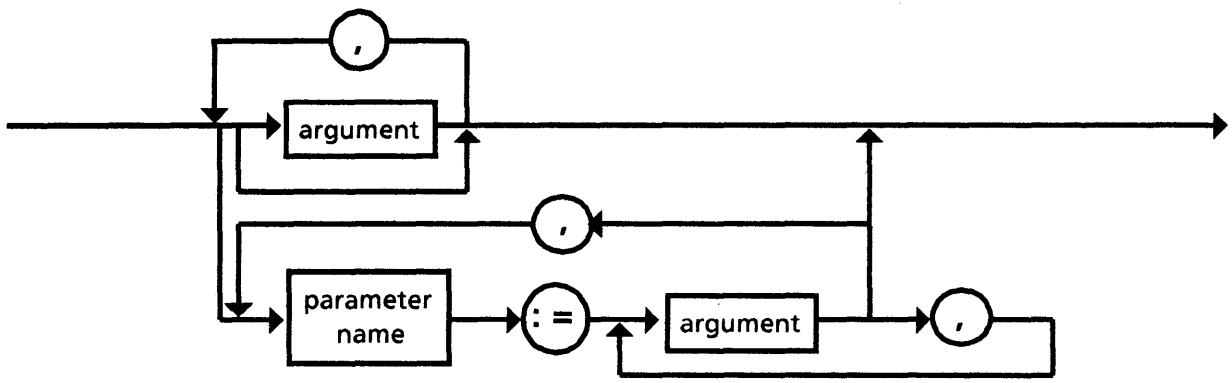
(for value parameters)

(for function name parameters)

(for procedure name parameters)

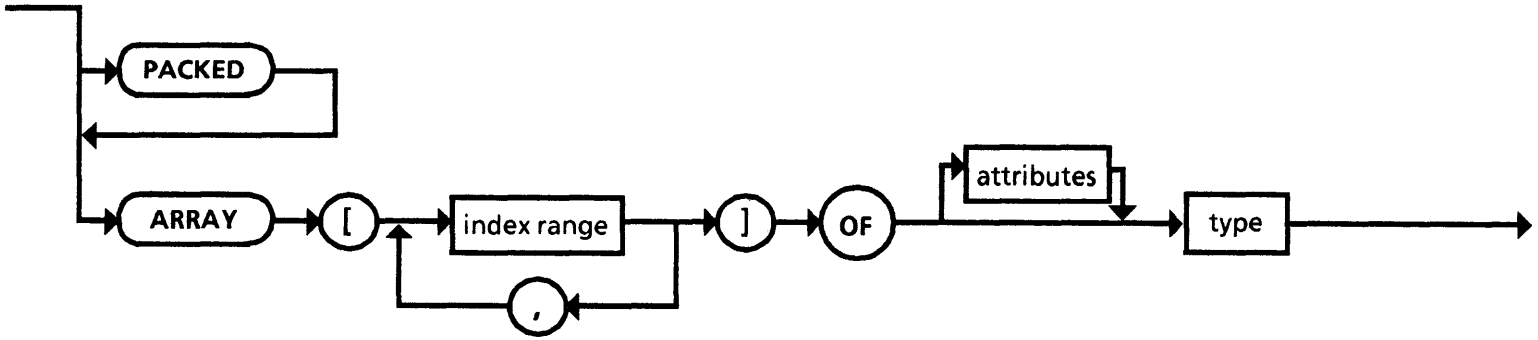
Note: The function name or procedure name may be a function call to the ARGUMENT function, whose first argument is an appropriate procedural parameter with the LIST attribute.

Argument List



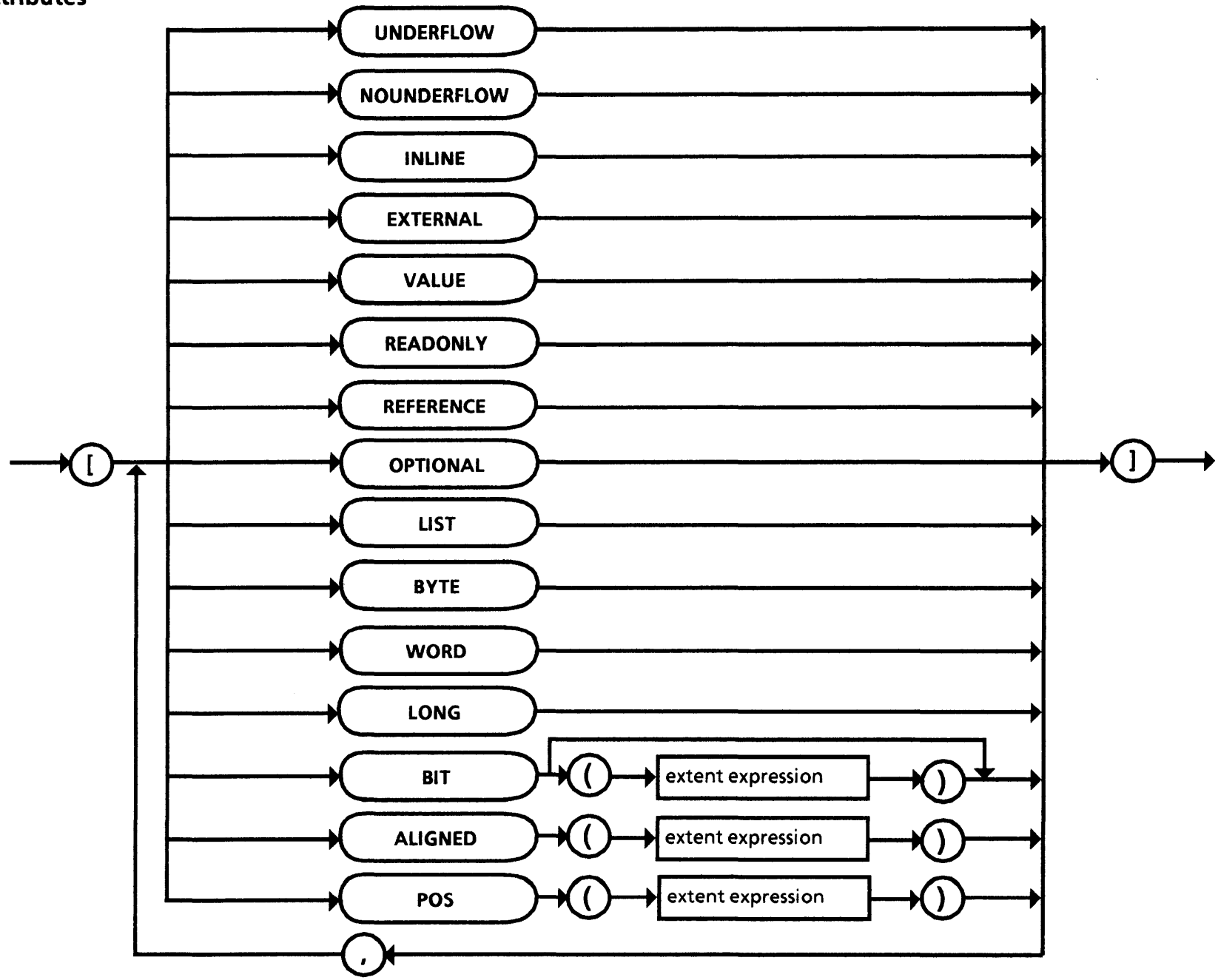
Note: A completely empty argument list is not allowed.

Array Type Definition

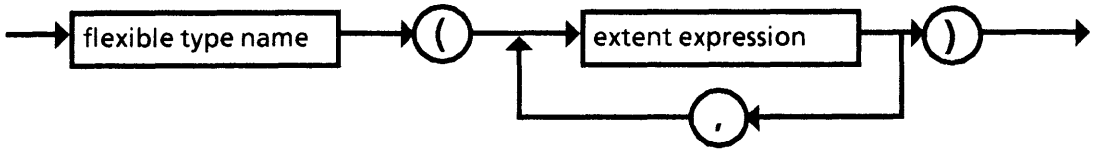


Assignment Statement

Attributes

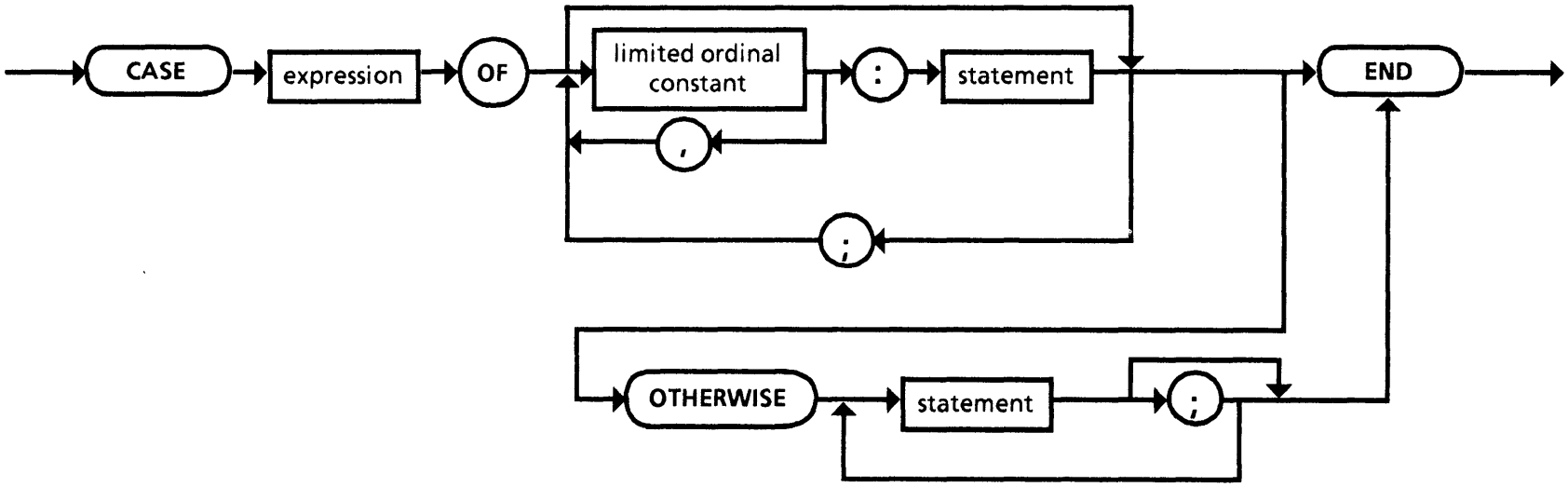

UNDERFLOW
NOUNDERFLOW
INLINE
EXTERNAL
VALUE
READONLY
REFERENCE
[
OPTIONAL
]
LIST
BYTE
WORD
LONG
BIT
(
extent expression
)
ALIGNED
(
extent expression
)
POS
(
extent expression
)
,

Bound Flexible Type

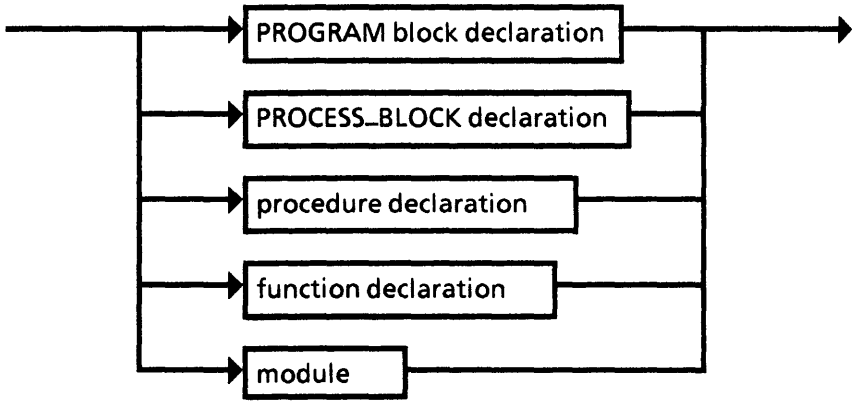


Buffer Variable Reference

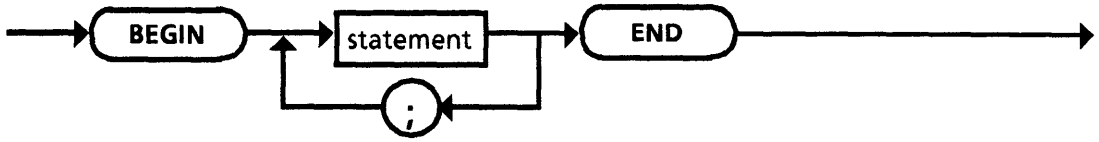
Case Statement



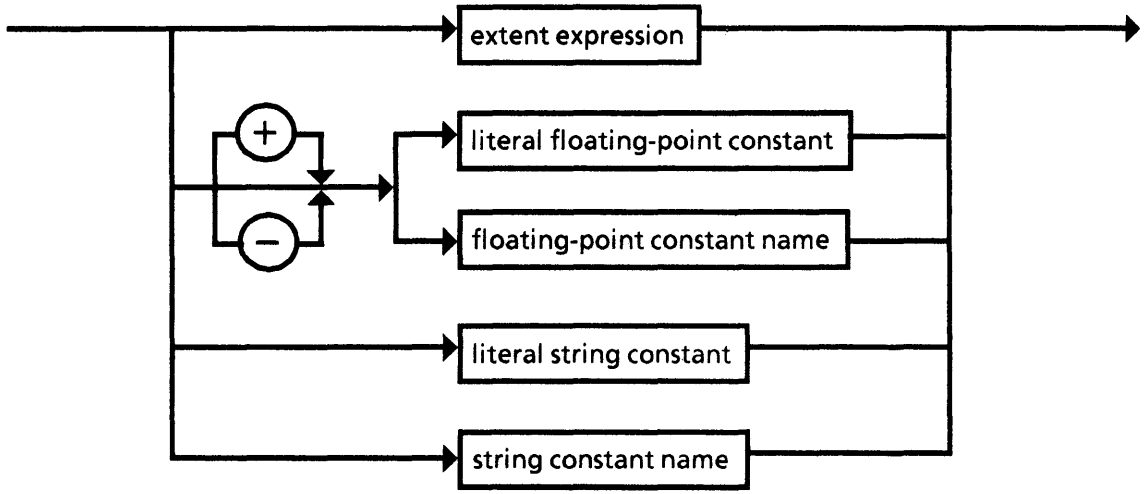
Compilation Unit



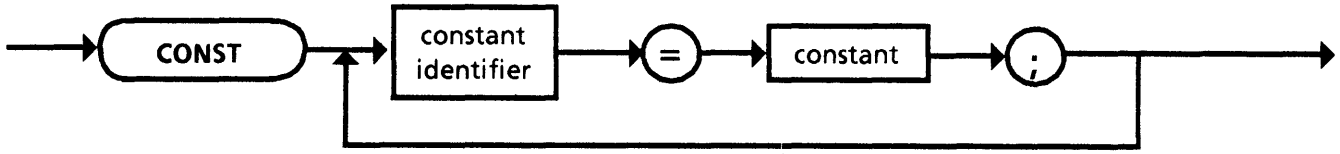
Compound Statement



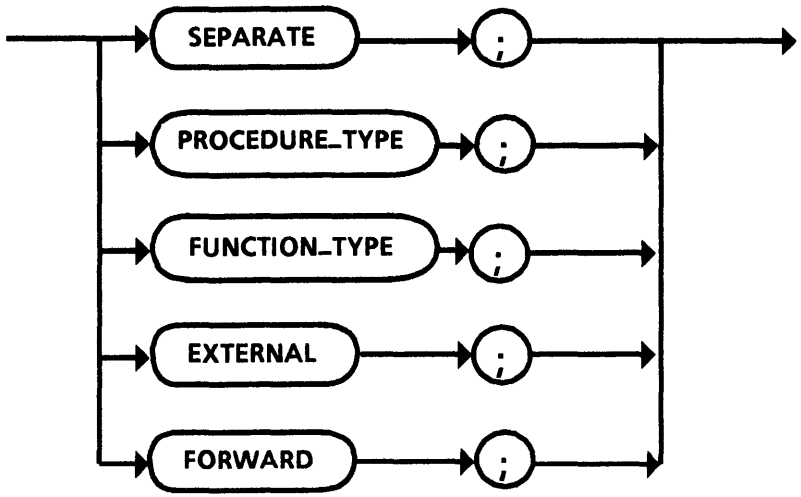
Constant



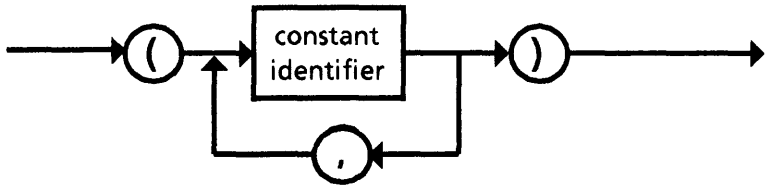
Constant Declaration



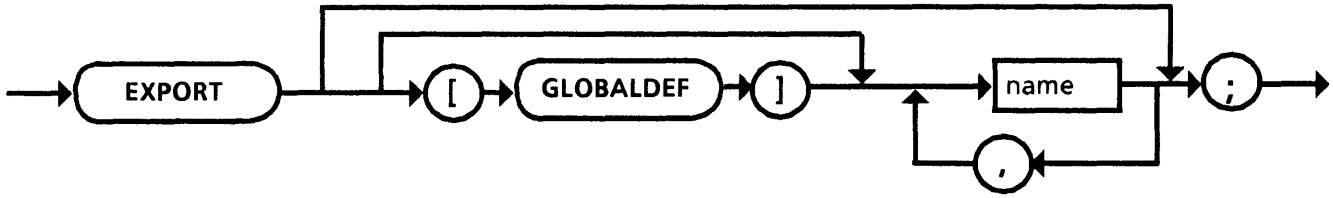
Directive



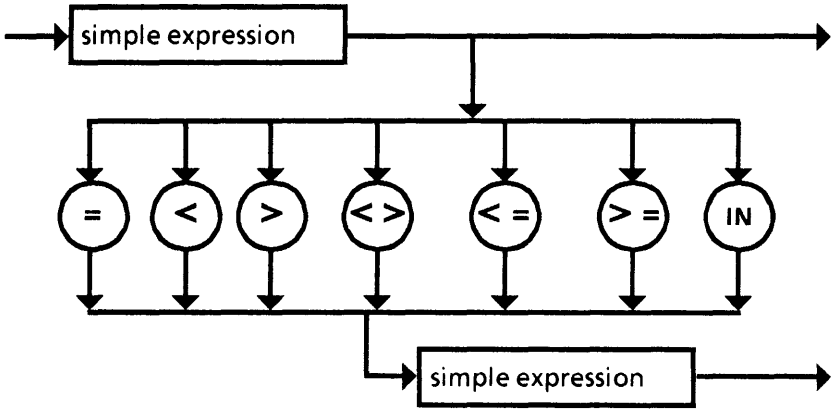
Enumerated Type Definition



Export Header



Expression

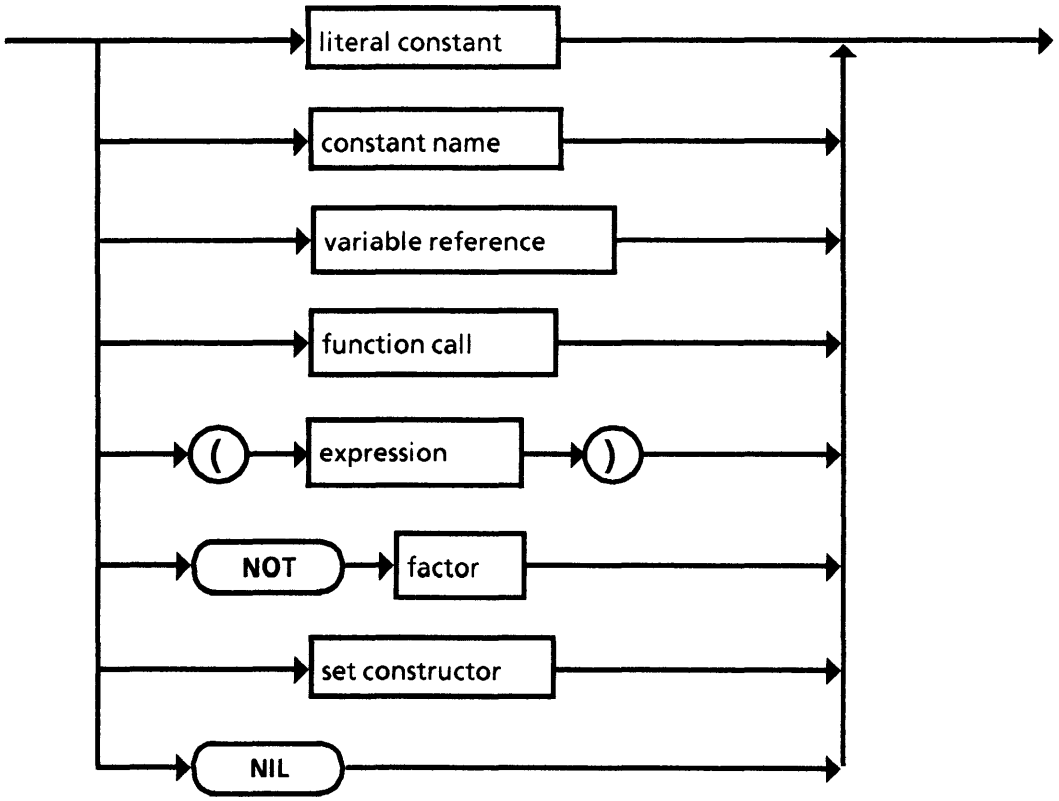


Extent Expression

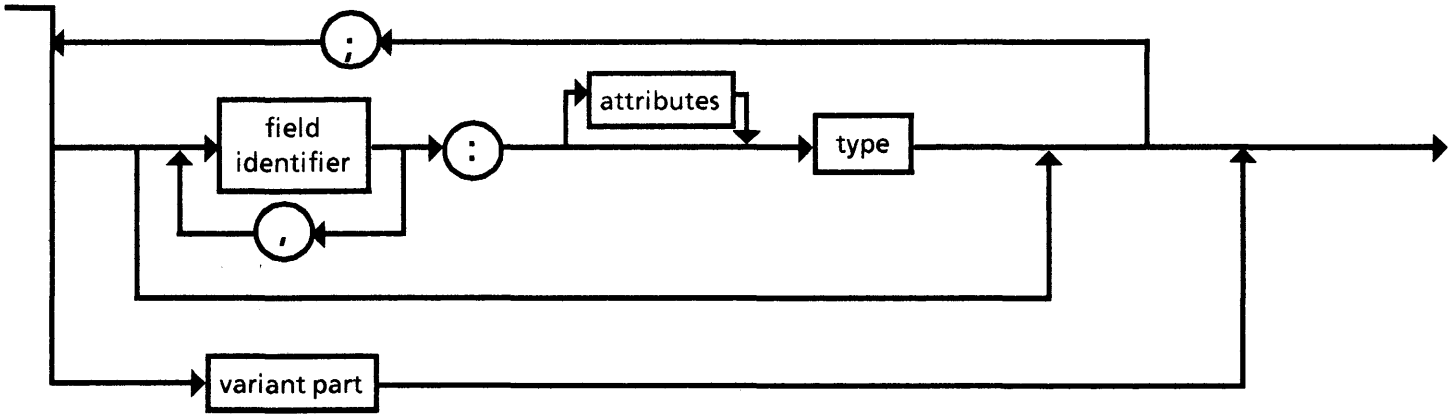
Notes: 1. Extent expressions have ordinal constant operands and produce ordinal results.

2. Only the following set of operations is permitted:
   - The dyadic +, −, *, AND, OR, DIV, and MOD operators (string concatenations with + are not allowed).
   - The monadic operators +, −, and NOT.
   - Relational operators (for example, <).
   - The functions ODD, ORD, PRED, SQR, ABS, CHR, SUCC, and XOR.

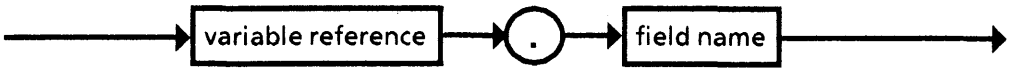
Factor


literal constant
constant name
variable reference
function call
(
expression
)
NOT
factor
set constructor
NIL

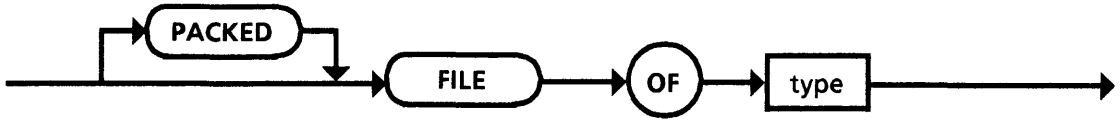
Field List



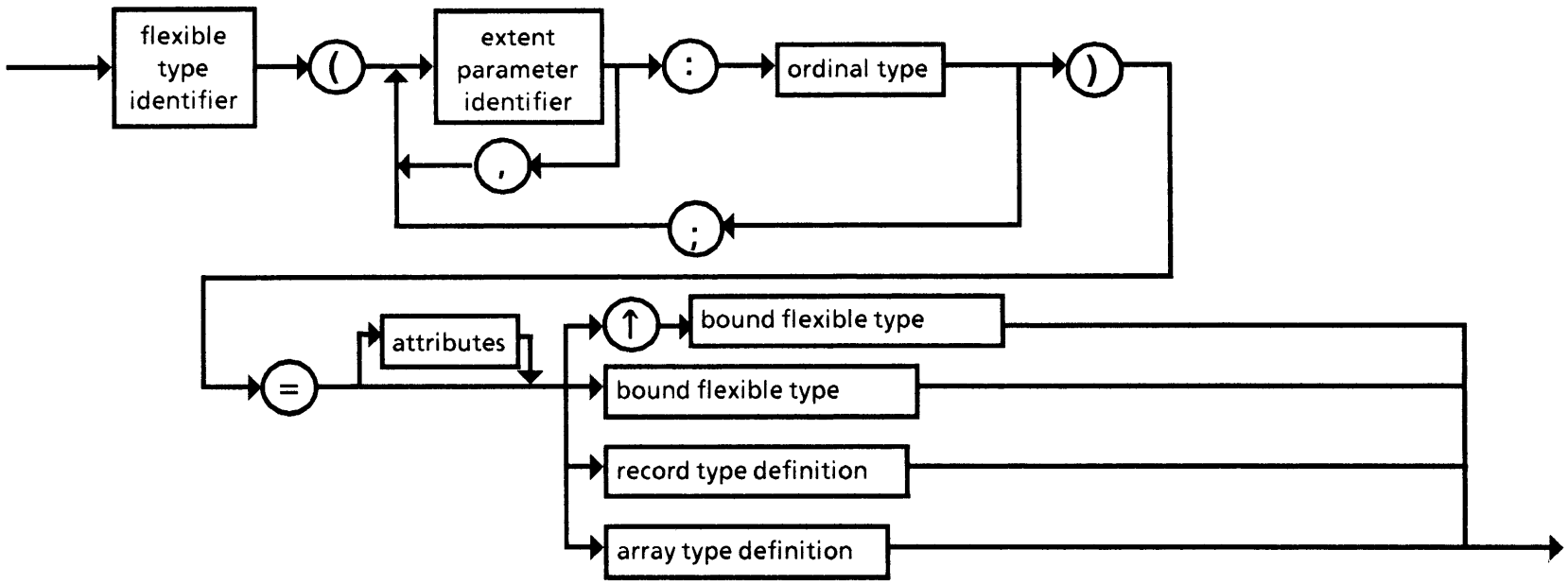
Field Reference



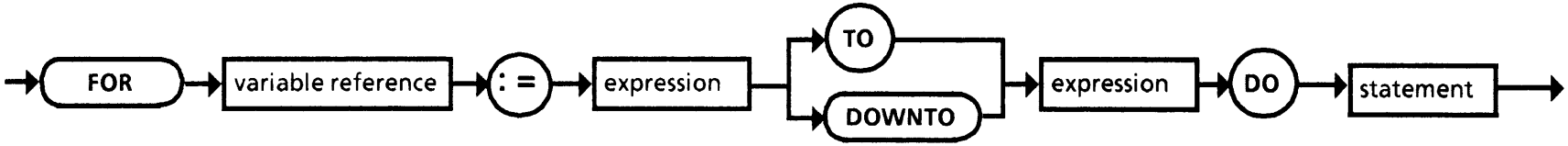
File Type Definition



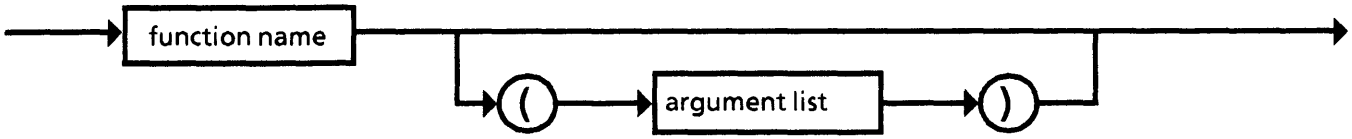
Flexible Type Definition



FOR Statement

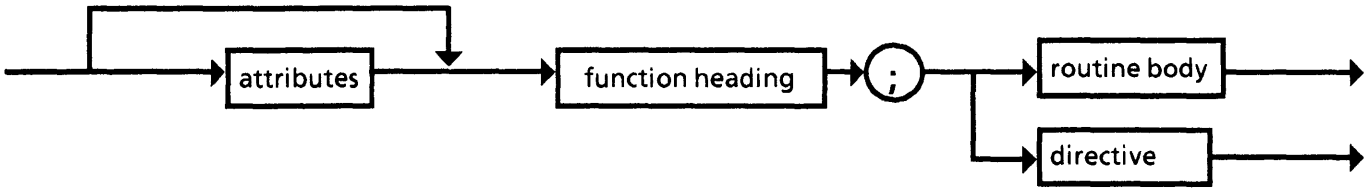


Function Call

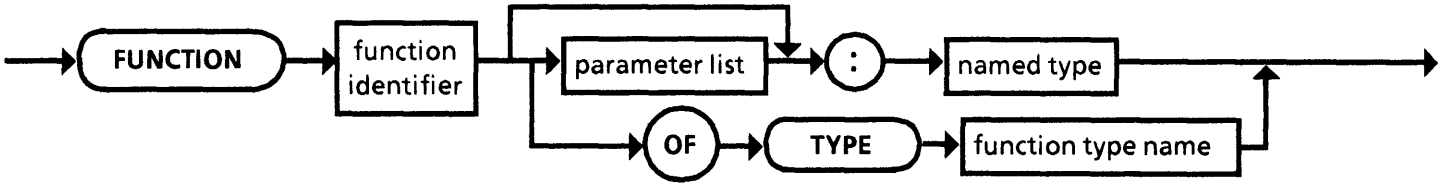


Note: The function name may be a function call to the ARGUMENT function, whose first argument is an appropriate procedural parameter with the LIST attribute.

Function Declaration

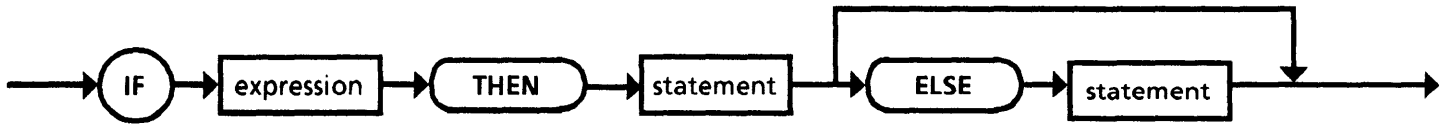


Function Heading

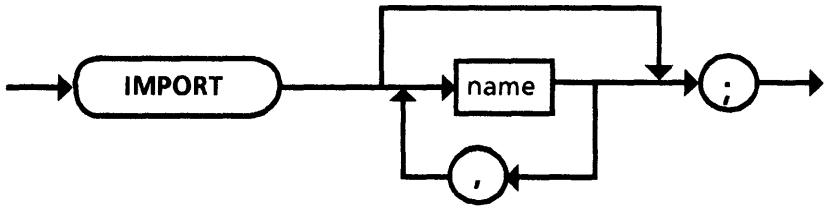


GOTO Statement

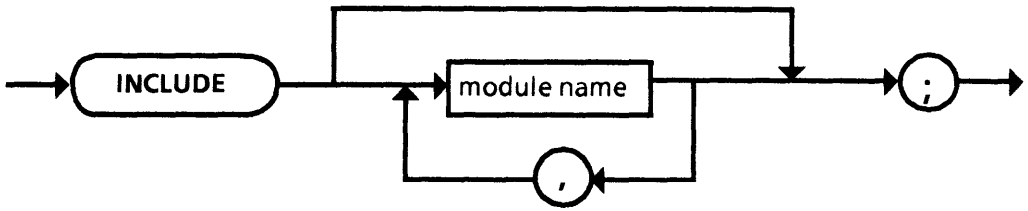
IF Statement



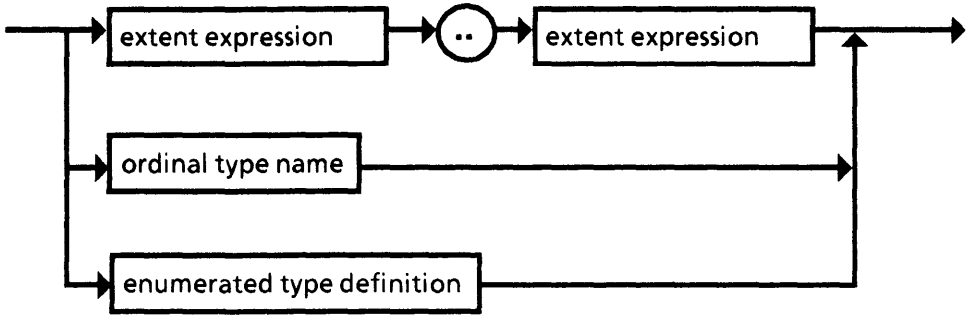
Import Header



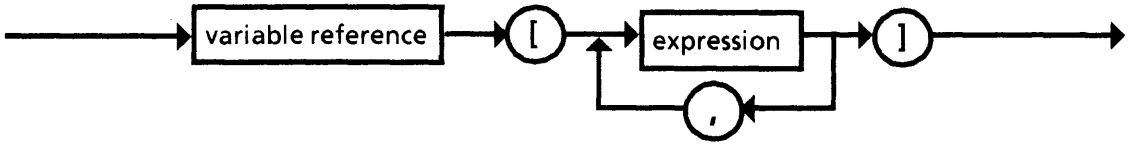
Include Header



Index Range

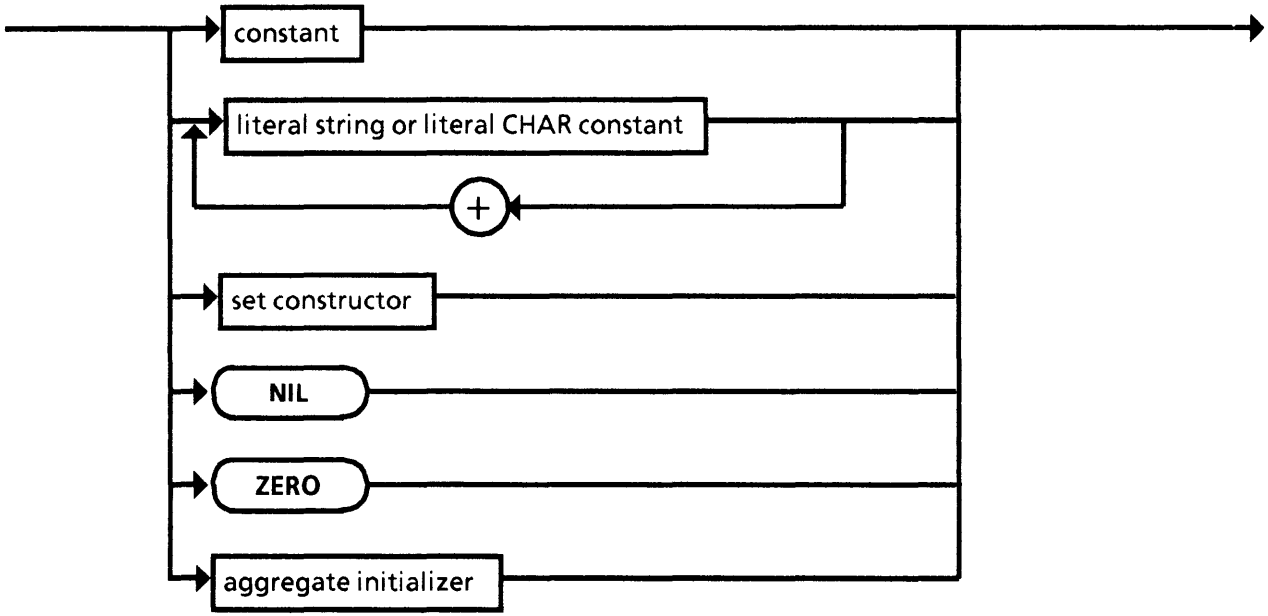


Indexed Variable Reference

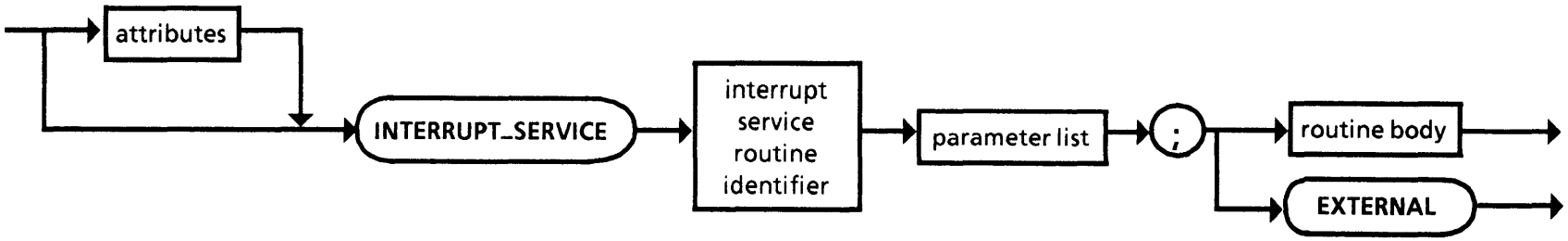


Indirect Variable Reference

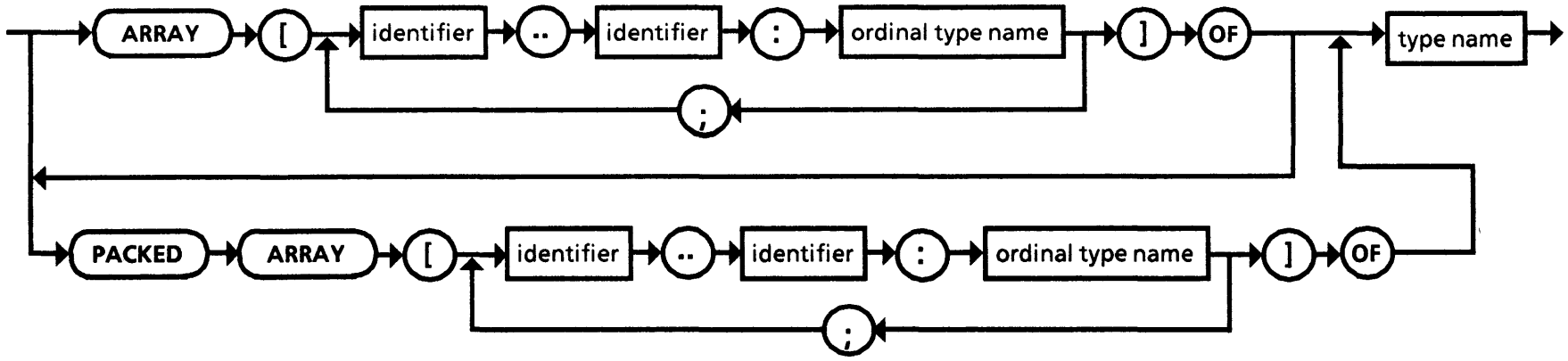
Initializer



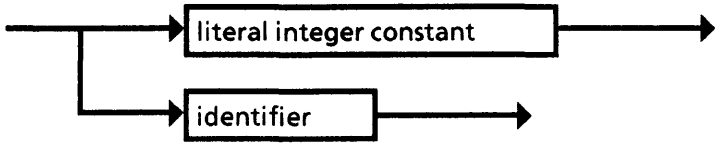
Interrupt Service Routine Declaration



ISO Conformant Type

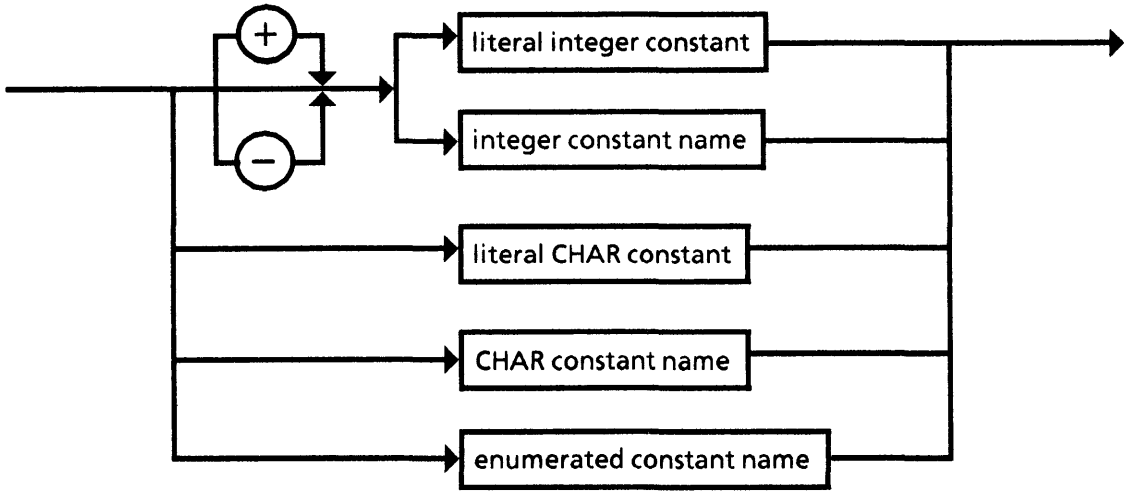


Label

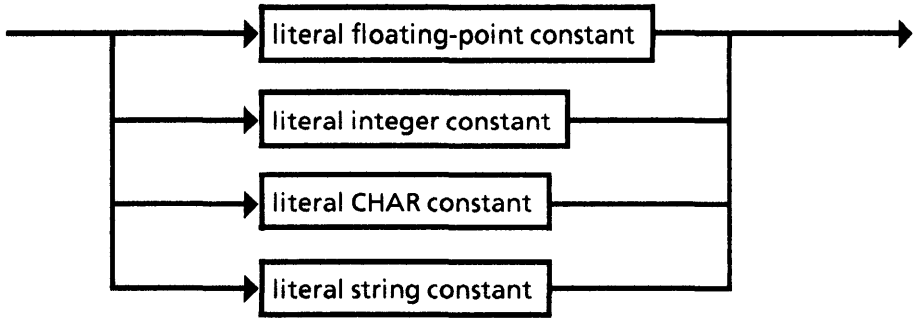


Note: The literal integer constant must be an unsigned decimal integer.

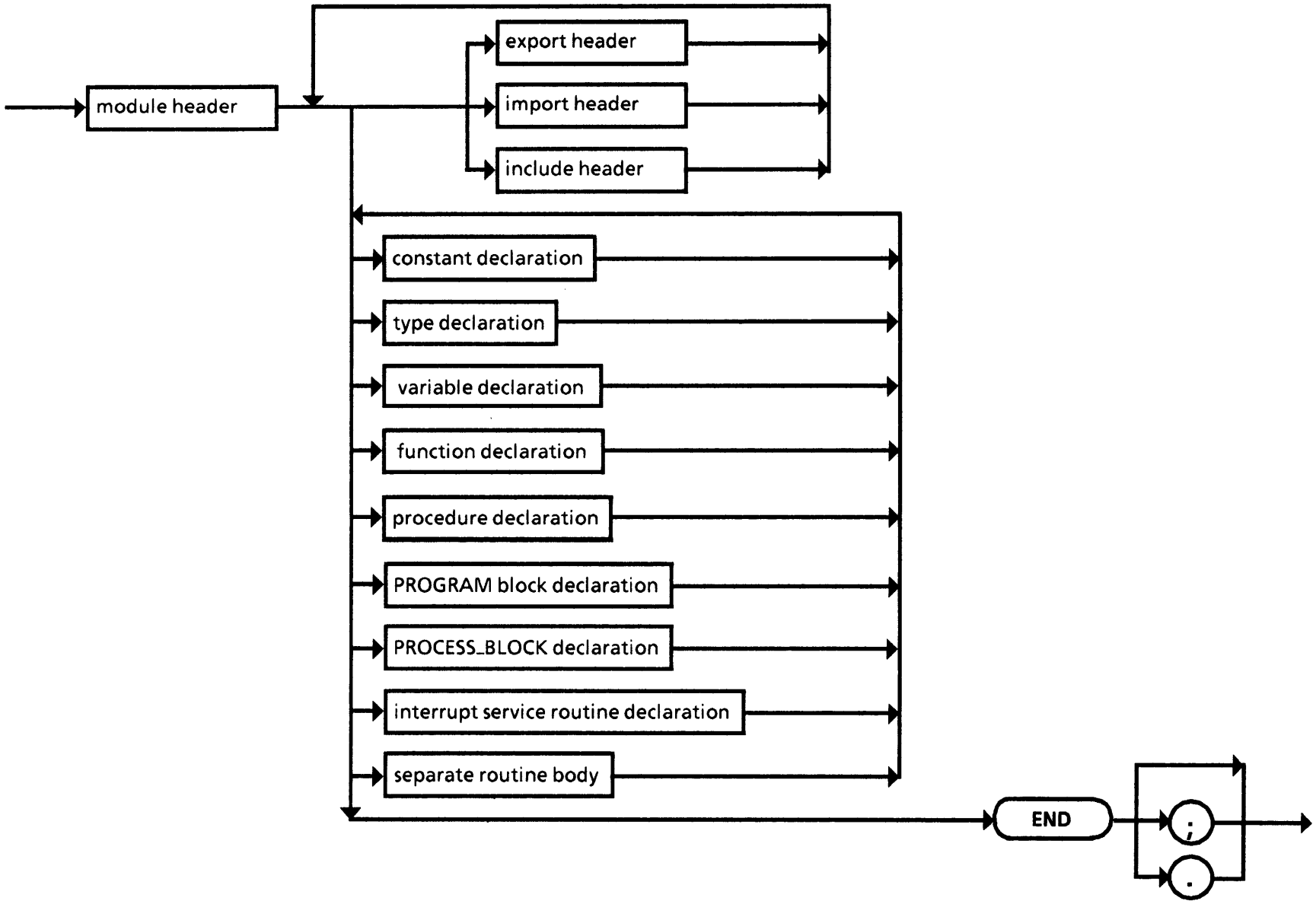
Limited Ordinal Constant



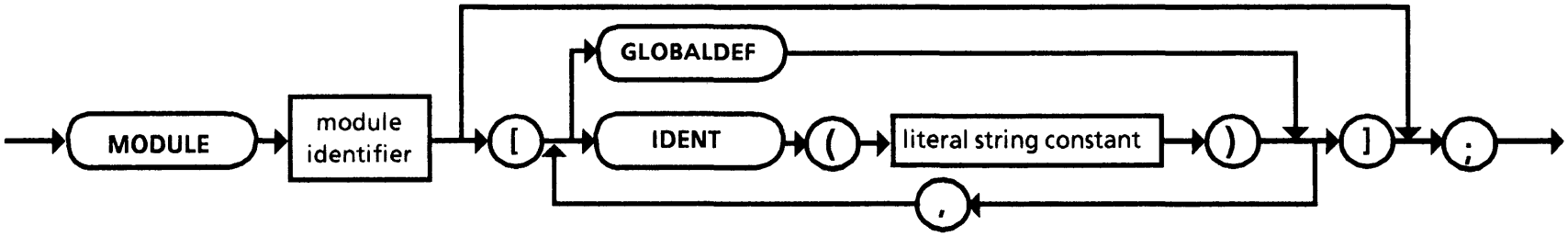
Literal Constant



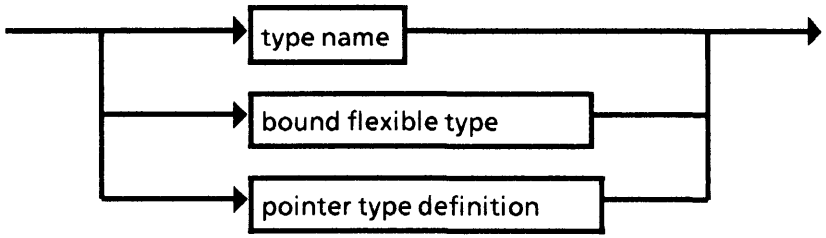
Module


module header
export header
import header
include header
constant declaration
type declaration
variable declaration
function declaration
procedure declaration
PROGRAM block declaration
PROCESS_BLOCK declaration
interrupt service routine declaration
separate routine body
END
;
.

Module Header

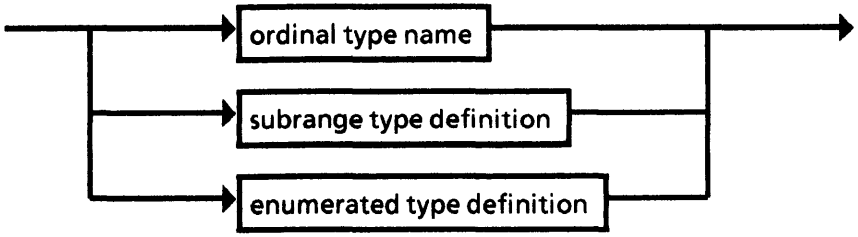


Named Type

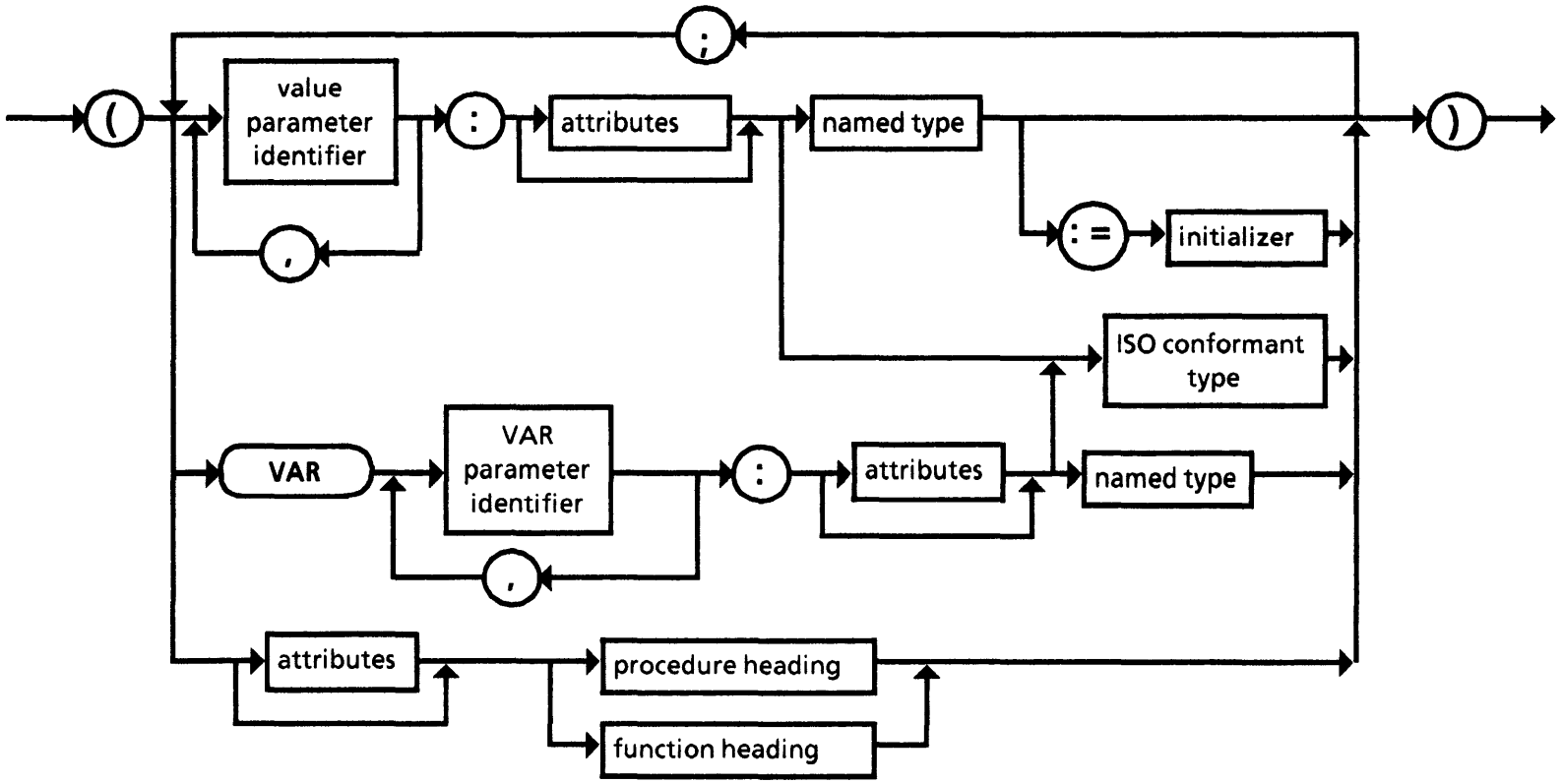


Null Statement

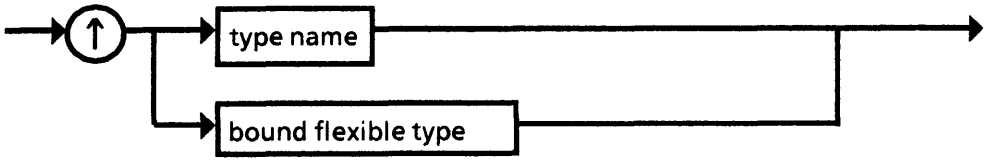
Ordinal Type



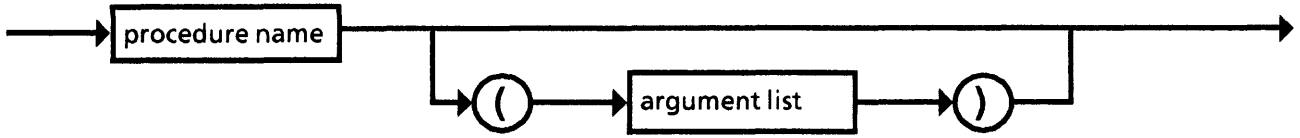
Parameter List



Pointer Type Definition

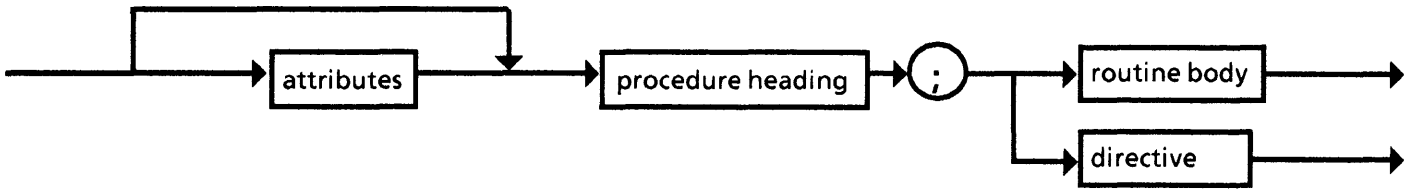


Procedure Call

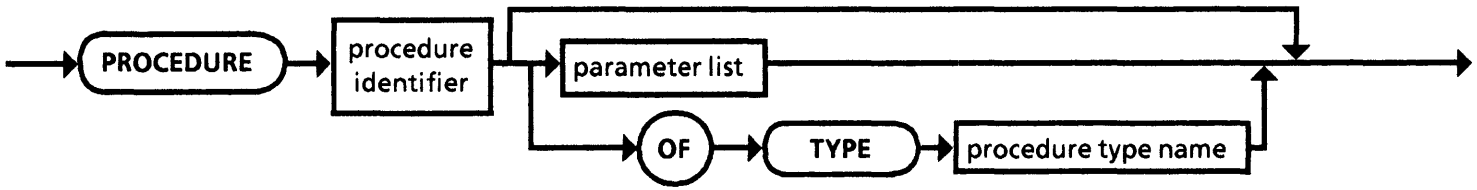


Note: The procedure name may be a function call to the ARGUMENT function, whose first argument is an appropriate procedural parameter with the LIST attribute.

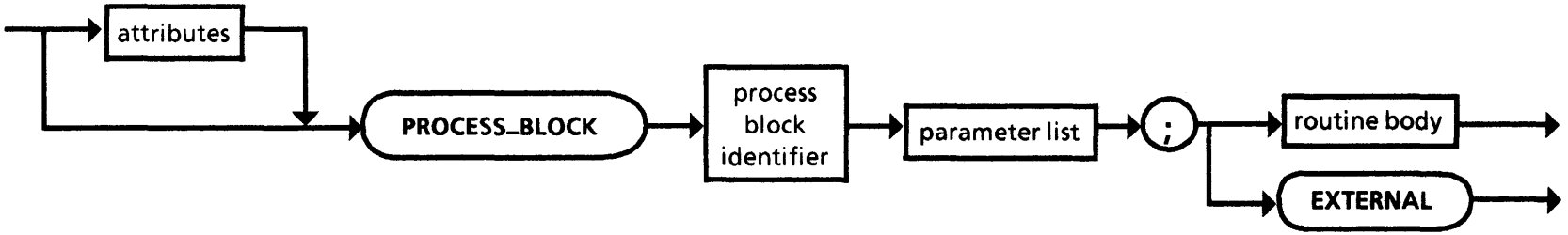
Procedure Declaration



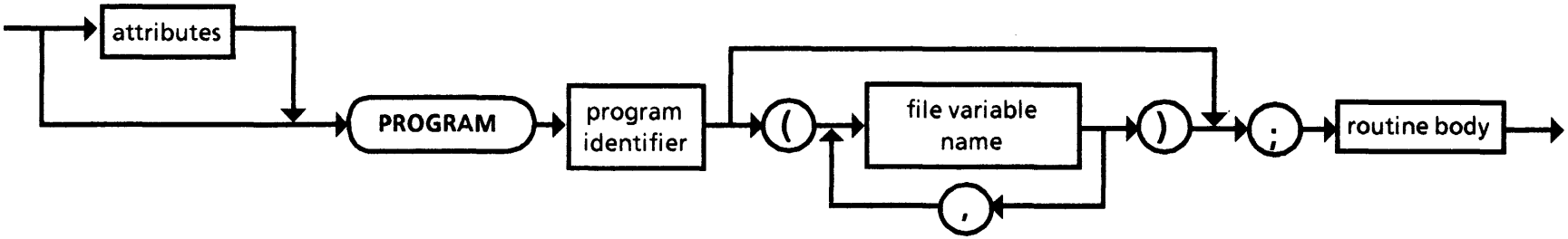
Procedure Heading



PROCESS_BLOCK Declaration



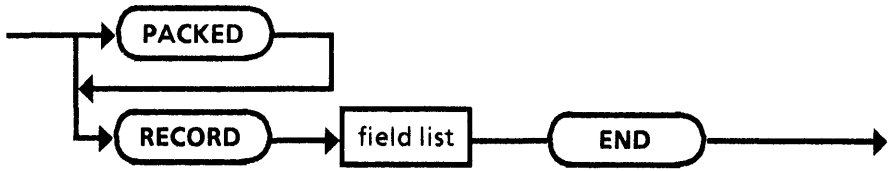
PROGRAM Block Declaration



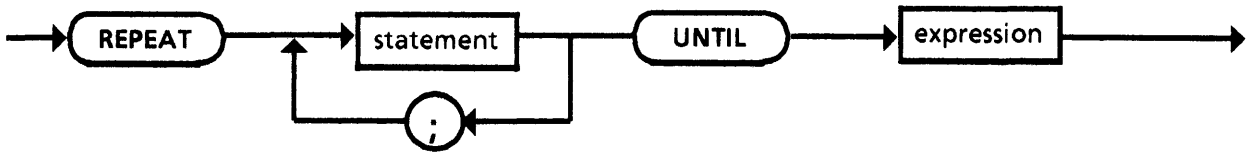
Pseudo Variable Reference

Note: The function call is an invocation of the SUBSTR function with a variable reference as its first argument, or the ARGUMENT function with a VAR parameter as its first argument.

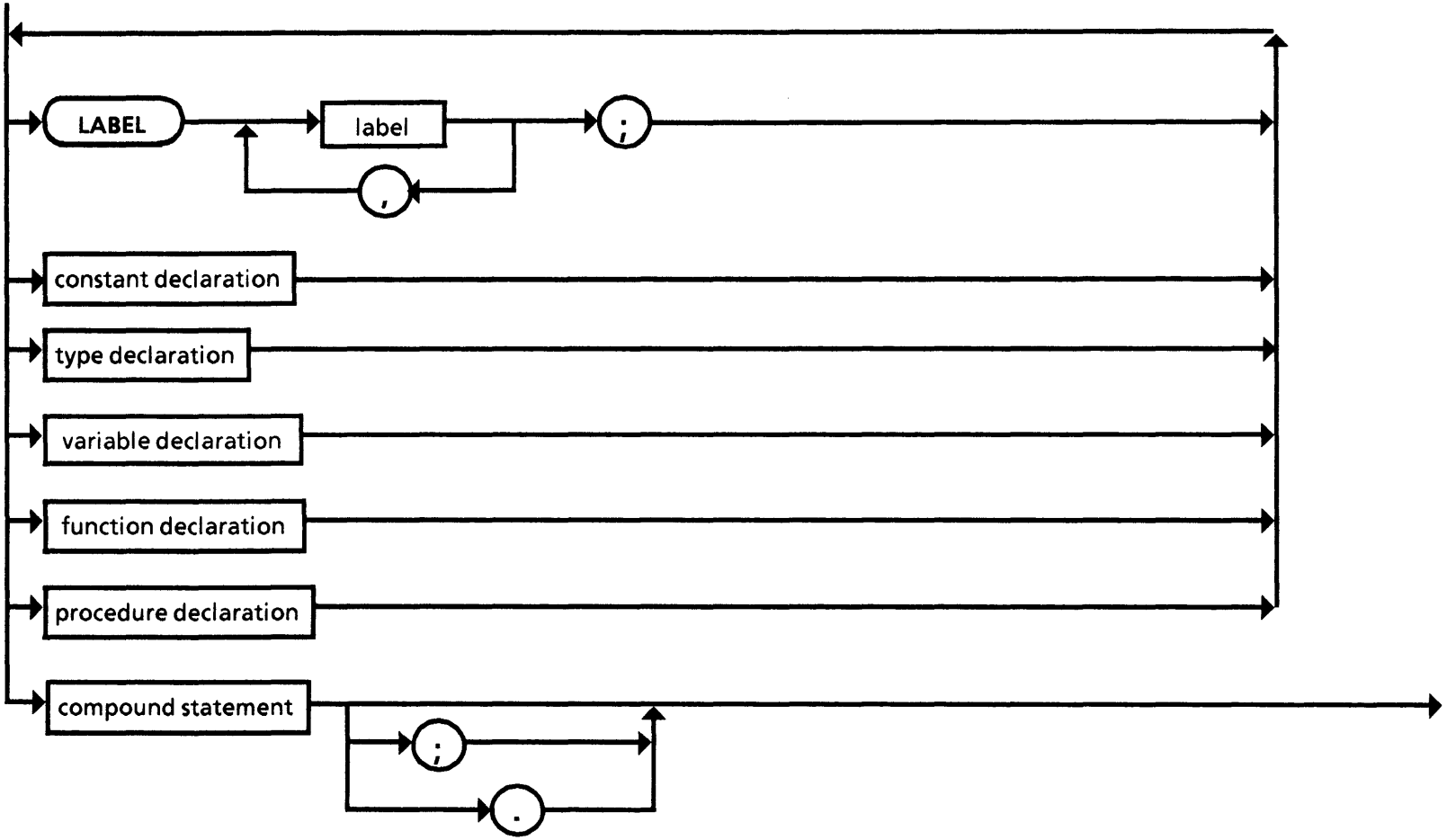
Record Type Definition



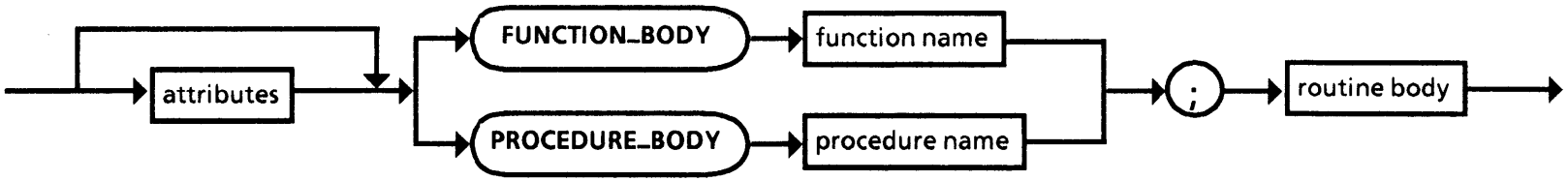
REPEAT Statement



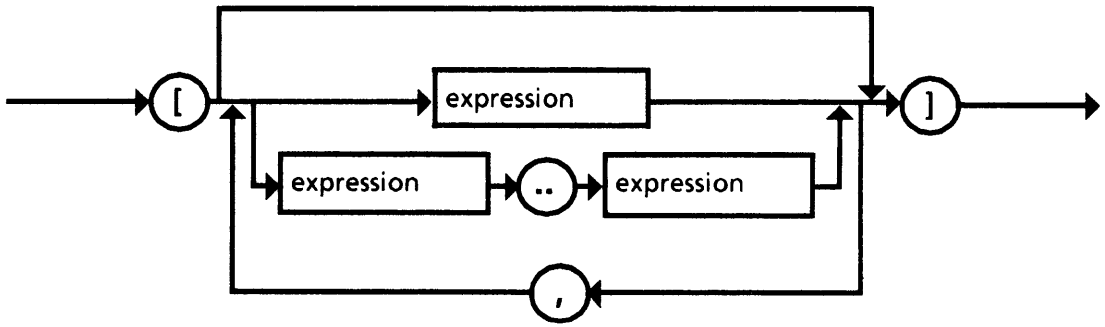
Routine Body



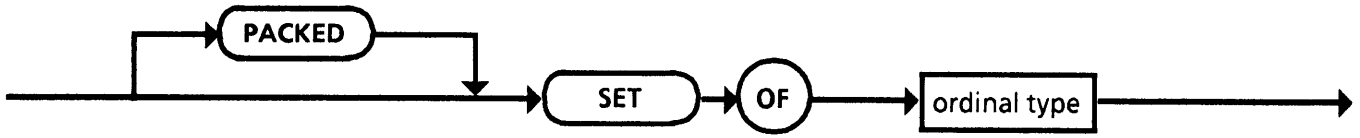
Separate Routine Body



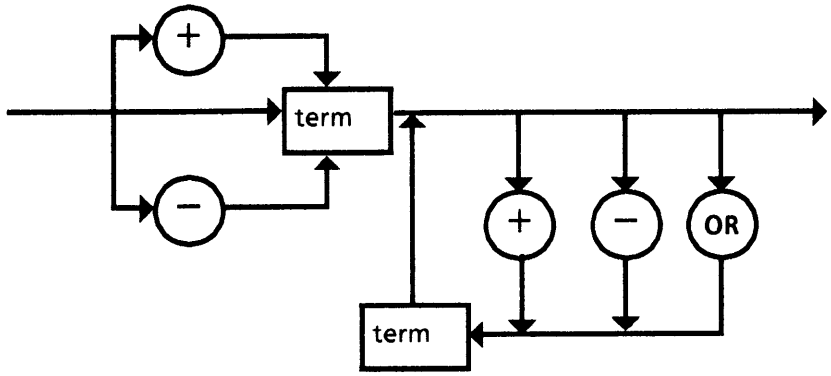
Set Constructor



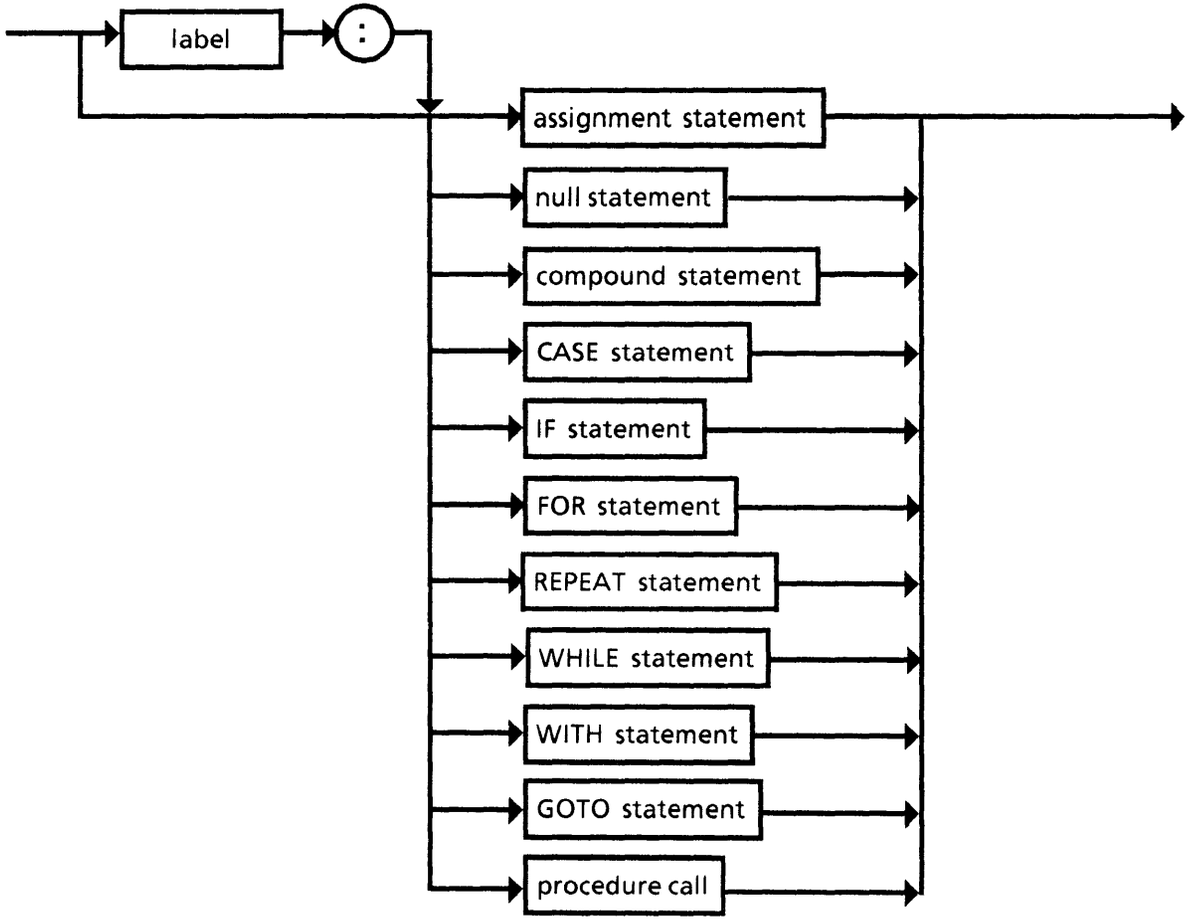
Set Type Definition



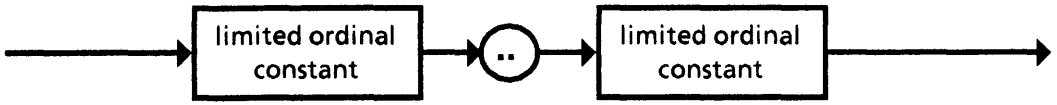
Simple Expression



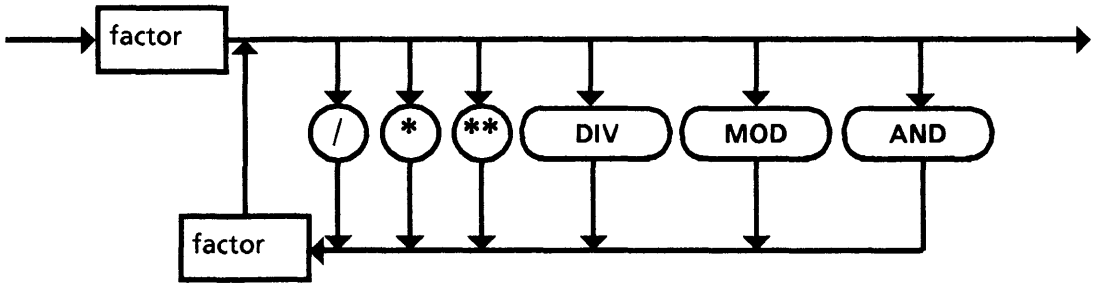
Statement


label
:
assignment statement
null statement
compound statement
CASE statement
IF statement
FOR statement
REPEAT statement
WHILE statement
WITH statement
GOTO statement
procedure call

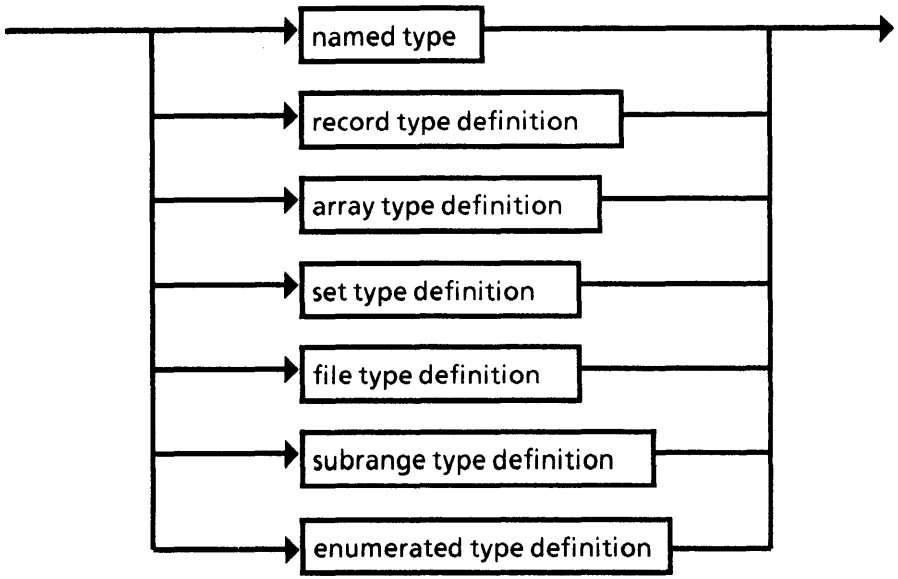
Subrange Type Definition



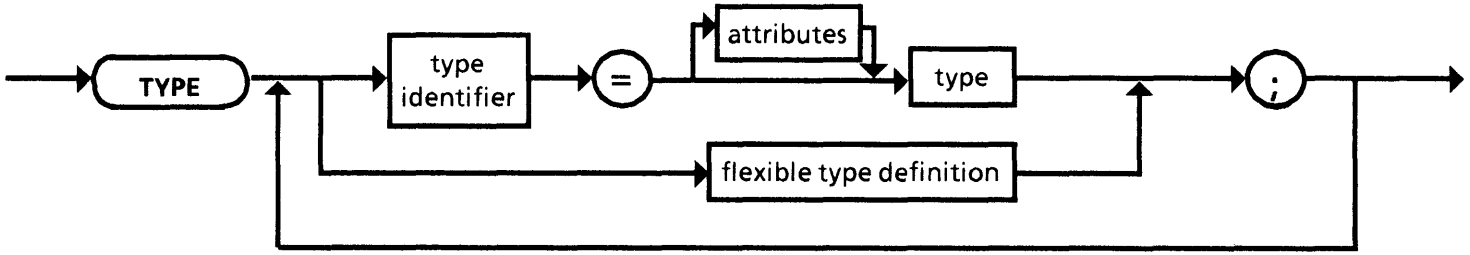
Term



Type

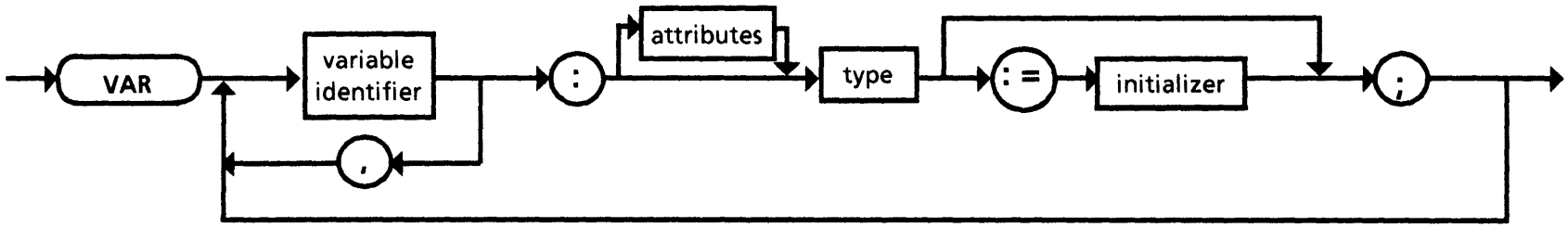


Type Declaration

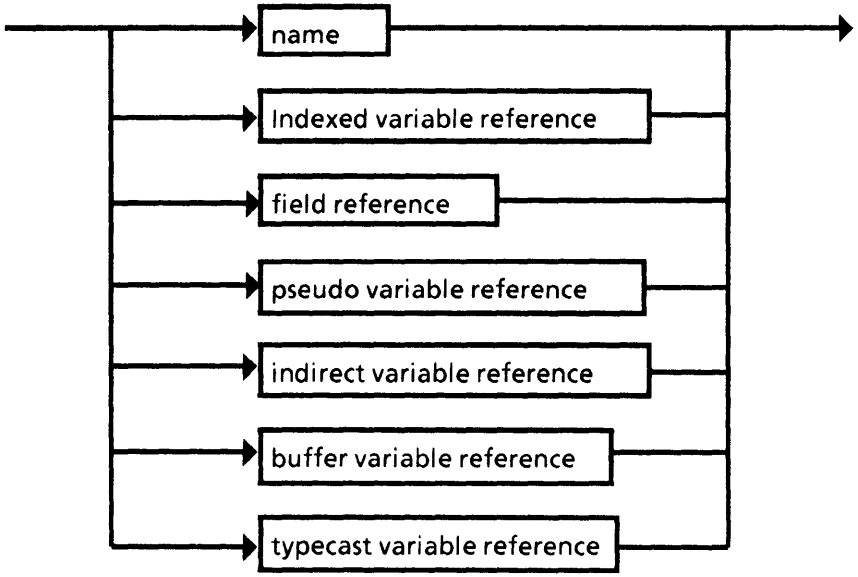


Typecast Variable Reference

Variable Declaration

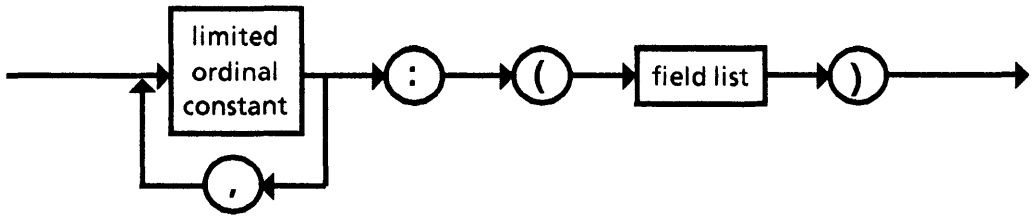


Variable Reference

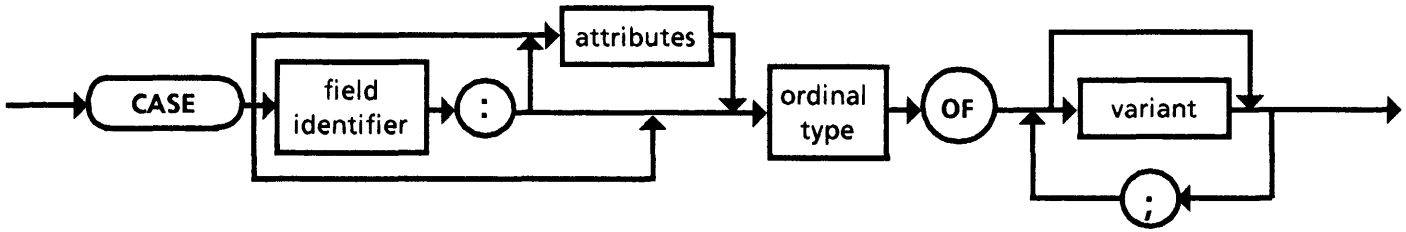


Note: The name is the name of a variable (VAR declaration), value parameter, VAR parameter, a name established by a WITH statement as a variable name, or a function name (left-hand side of assignment within the function).

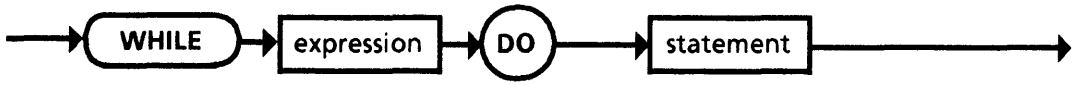
Variant



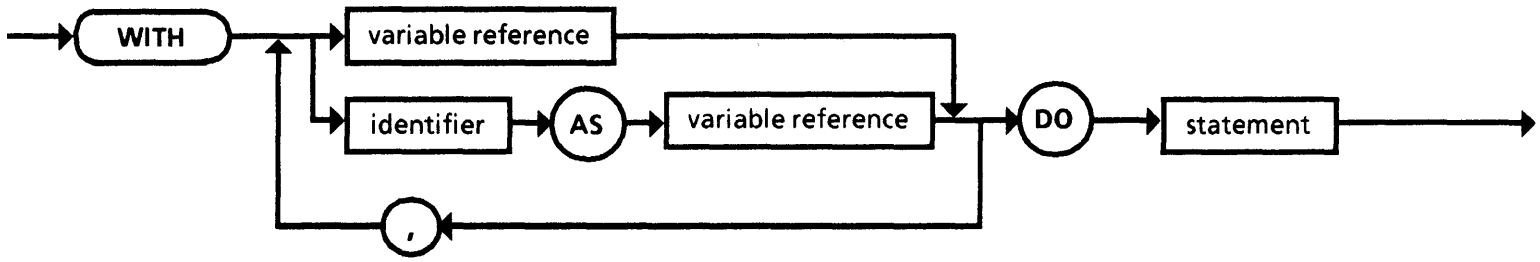
Variant Part



WHILE Statement



WITH Statement



Appendix C
Call Formats

This appendix lists the call formats of the procedures and functions available in VAXELN Pascal.

Chapter 1, "Notation and Lexical Elements," contains an explanation of the conventions used in these call formats. As explained in the conventions, predeclared routines must be called as documented. In particular, the call formats for VAXELN kernel procedures use the conventions listed in Chapter 1, unless the name is prefixed with KER$.

Note that some of the routines are provided with VAXELN but are not predeclared. They are used in programs by including particular modules from the RTLOBJECT library in the compilation. In Table C-1, they can be identified by the statement "include module *name*" in the routine's description. For these routines, the actual names and positions of the parameters are shown in the call format, and the usual VAXELN calling rules apply.

Table C-1. VAXELN Pascal Procedures and Functions

Call Format	Meaning
ABS(expression)	Return absolute value of *expression*.
ACCEPT_CIRCUIT(myport, FULL_ERROR := boolean, CONNECT := connect_port, ACCEPT_DATA := varying_string, CONNECT_DATA := varying_string, STATUS := integer_variable)	Establish circuit between *myport* and originator of connection request; if the full error is disabled (FALSE, default), SEND will wait implicitly when the partner port is full (otherwise, an error status is returned by SEND); the varying strings supply optional data to the originator (accept) or receive data (connect). The optional *connect_port* specifies a different port on which to make the actual connection.
ADD_INTERLOCKED(delta,word_argument)	Return sum of the integer *word_argument* and *delta* in *word_argument* and the function result 1 if the sum is positive, 0 if the sum is zero, and −1 if the sum is negative.
ADDRESS(variable)	Return pointer to *variable*.
KER$ALLOCATE_MAP(*status*, register, number, count, device_object, *spt_address*)	Allocate *count* UNIBUS or QBUS map registers, returning first register *number*, for DEVICE value *device_object*, and return pointer to first (↑ ANYTYPE) in *register*. The optional INTEGER variable *status* receives the completion status, and *spt_address* receives an ANYTYPE pointer to the system page table base. Include module $KERNEL.
ALLOCATE_MEMORY(mempointer, size, VIRTUAL := address, PHYSICAL := integer, STATUS := integer_variable)	Allocate *size* bytes of memory, beginning at virtual *address*, and return ↑ ANYTYPE in *mempointer*. Optionally, the *integer* can supply the starting physical address explicitly; the program must be in kernel mode for this use.
KER$ALLOCATE_PATH(*status*, register, number, dev)	Allocate UNIBUS buffered datapath for DEVICE value *dev*, and return pointer to datapath register (↑ ANYTYPE) in *register* and the datapath number in *number*. The optional INTEGER variable *status* receives the completion status. Include module $KERNEL.
ELN$ALLOCATE_STACK(stack_size, *status*)	Verify that the process has the *stack_size* requested; if not, allocate the stack space. The optional INTEGER variable *status* receives the completion status. Include module $STACK_UTILITY.
ARCTAN(expression)	Return angle in radians whose tangent is *expression*.
ARGUMENT(name,expression)	Refer to LIST parameter *name* in list position given by *expression*.
ARGUMENT_LIST_LENGTH(name)	Return number of arguments for LIST parameter *name*.

Table C-1. Continued

Call Format	Meaning
ASSERT(boolean_expression)	Raise exception or compiler error if *boolean_expression* is FALSE.
ELN$AUTH_ADD_USER(*status*, circuit, username, nodename, password, uic, userdata)	Add a new user record with *username*, *password*, and *uic* to the authorization database, to be authorized on *nodename*. The PORT value specifying the port connected in a circuit to the Authorization Service's AUTH$MAINTENANCE port is supplied by *circuit*, and *userdata* supplies an arbitrary string of user-specified data. The optional INTEGER variable *status* receives the completion status. Include module $AUTHORIZE_UTILITY.
ELN$AUTH_MODIFY_USER(*status*, circuit, username, nodename, new_fields, new_username, new_nodename, new_password, new_uic, new_userdata)	Modify an existing user record in the authorization database, uniquely identified by *username* and *nodename*. The PORT value specifying the port connected in a circuit to the Authorization Service's AUTH$MAINTENANCE port is supplied by *circuit*, and *new_fields* supplies a set that specifies which of the other fields are to be modified. The optional INTEGER variable *status* receives the completion status. Include module $AUTHORIZE_UTILITY.
ELN$AUTH_REMOVE_USER(*status*, circuit, username, nodename)	Remove an existing user record from the authorization database, uniquely identified by *username* and *nodename*. The PORT value specifying the port connected in a circuit to the Authorization Service's AUTH$MAINTENANCE port is supplied by *circuit*. The optional INTEGER variable *status* receives the completion status. Include module $AUTHORIZE_UTILITY.
ELN$AUTH_SHOW_USER(*status*, circuit, username, nodename, show_user)	Return authorization database information for the specified user or users identified by *username* and *nodename*. The PORT value specifying the port connected in a circuit to the Authorization Service's AUTH$MAINTENANCE port is supplied by *circuit*, and *show_user* supplies a procedure name that identifies a user-specified routine to be invoked with the values of the specified user record or all the records in the authorization data file. The optional INTEGER variable *status* receives the completion status. Include module $AUTHORIZE_UTILITY.

Table C-1. Continued

Call Format	Meaning
ELN$AXV_INITIALIZE(device_name, identifier, maximum_values, *clock_start_enable*, *external_start_enable*, *re_initialize*, *use_polling*, *status*)	Ready *device_name* ADV or AXV device for input and/or output and create all needed data structures. The variable *identifier* receives a longword identifier to be used to identify the device in subsequent calls. The maximum number of values that can be read from the device in a single call to AXV_READ is supplied by *maximum_values*. The optional arguments *clock_start_enable*, *external_start_enable*, *re_initialize*, and *use_polling* are BOOLEAN expressions. The optional INTEGER variable *status* receives the completion status. Include module $AXV_UTILITY.
ELN$AXV_READ(identifier, start_channel, end_channel, reads_per_channel, data_array_ptr, *kwv_ident*, *gain_array*, *status*)	Read *reads_per_channel* analog data from ADV or AXV device *identifier*, from *start_channel* to *end_channel*, and convert it to binary form. The variable *data_array_ptr* receives the address of an array containing converted data from the device. The optional *kwv_ident* supplies the identifier of a KWV real-time clock device. The optional *gain_array* supplies the gain to be used in the data conversion for each channel to be read. The optional INTEGER variable *status* receives the completion status. Include module $AXV_UTILITY.
ELN$AXV_WRITE(identifier, dac_channel, value, *status*)	Write *value* to an analog-to digital conversion output register *dac_channel* on AXV device *identifier*. The optional INTEGER variable *status* receives the completion status. Include module $AXV_UTILITY.
BIN(expression,*length*,*digits*)	Return string giving binary notation for *expression*, with optionally specified *length* and minimum number of significant *digits*.
ELN$CANCEL_EXIT_HANDLER(exit_handler, *exit_context*)	Cancel an exit handler, identified by *exit_handler* and *exit_context*, previously enabled by DECLARE_EXIT_HANDLER. Include module $EXIT_UTILITY.
CHR(ordinal_expression)	Return character whose ordinal number is *ordinal_expression*.
CLEAR_EVENT(event, STATUS:=integer_variable)	Set state of *event* to EVENT$CLEARED.
CLOSE(file)	Close *file*.

Table C-1. Continued

Call Format	Meaning
CONNECT_CIRCUIT(myport, DESTINATION_PORT := port, DESTINATION_NAME := name, FULL_ERROR := boolean_expression, CONNECT_DATA := varying_string, ACCEPT_DATA := varying_string, STATUS := integer_variable)	Request circuit connection between *myport* and destination *name* or *port*; if the full error is disabled (FALSE, default), SEND will wait implicitly when the partner port is full (otherwise, an error status is returned by SEND); the varying strings supply optional data to the destination (connect) or receive data when the destination accepts. (The destination must be specified either by NAME or PORT value.)
CONVERT(type,expression)	Return *expression* converted to *type*.
ELN$COPY_FILE(source_file, destination_file, *status*, *source_file_error*, *block_mode*, *count*, *resultant_source_file*, *resultant_destination_file*)	Copy *source_file* to *destination_file*. The optional INTEGER variable *status* receives the completion status. The optional *source_file_error* is TRUE if an error exists in *source_file*, FALSE if an error exists in *destination_file*. The optional *block_mode* is TRUE if block mode, FALSE if record mode. The optional INTEGER variable *count* specifies the number of blocks or records. The optional *resultant_source_file* and *resultant_destination_file* return the resultant filename strings. Include module $FILE_UTILITY.
COS(expression)	Return cosine of *expression* (angle in radians).
CREATE_AREA(area_variable, data_pointer, area_name, VIRTUAL := virtual_address, STATUS := integer_variable)	Create a new area or map an existing area of memory with a unique *area_name* and return the AREA value in *area_variable*. The variable *data_pointer* receives a pointer to the beginning of the allocated memory. The optional *virtual_address* specifies the exact P0 base address.
CREATE_DEVICE(device_name, device_variable, VECTOR_NUMBER := integer, SERVICE_ROUTINE := isr, REGION := rpointer, REGISTERS := regpointer, ADAPTER_REGISTERS := adpointer, VECTOR := vpointer, PRIORITY := integer_variable, POWERFAIL_ROUTINE := isr, STATUS := integer_variable)	Connect to *device_name* interrupt and return the DEVICE value in *device_variable*. The vector number is an integer from 1 (default) to 128. Interrupt service routines (*isrs*) can be supplied for interrupt and power recovery handling. The variable *rpointer* receives a pointer to the communication region; *regpointer* and *adpointer* receive pointers to the first device control register and first adapter control register, respectively; *vpointer* receives a pointer to the interrupt vector. Integer variables receive the interrupt priority level of the device and the completion status of the procedure. The *device_variable* can be a scalar DEVICE variable or an ARRAY[0..*n*] OF DEVICE, where $n \leq 15$.

Table C-1. Continued

Call Format	Meaning
ELN$CREATE_DIRECTORY(directory_name, *status*, *owner*, *resultant_directory_name*)	Create the specified directory (up to 255 characters in the file specification) and return the status in the optional INTEGER variable *status*. The optional *owner* supplies the user identification code of the owner of the file, and the optional *resultant_directory_name* returns the resultant filename string of the directory file. Include module $FILE_UTILITY.
CREATE_EVENT(event_variable, initial_state, STATUS := integer_variable)	Create an event with *initial_state* EVENT$SIGNALED or EVENT$CLEARED, and return EVENT value in *event_variable*.
CREATE_JOB(port_variable, program_name, argument-list, NOTIFY := exit_port_variable, STATUS := integer_variable)	Create a job running *program_name*, with optional arguments supplied to the program. The variable *port_variable* receives the value of the new job's job port. The variable *exit_port_variable* supplies a port that receives notification of the job's termination.
CREATE_MESSAGE(message_variable, data_pointer, STATUS := integer_variable)	Create a message with a data area suitable for the base type of the *data_pointer*, and return the MESSAGE value in *message_variable*.
ELN$CREATE_MUTEX(mutex, *status*)	Return new MUTEX value in *mutex* and the completion status in the optional INTEGER variable *status*. Include module $MUTEX.
CREATE_NAME(name_variable, string, port, TABLE := scope, STATUS := integer_variable)	Make *string* the name of *port*, with *scope* NAME$LOCAL (default), NAME$UNIVERSAL, or NAME$BOTH, and return the NAME value in *name_variable*.
CREATE_PORT(port_variable, LIMIT :=integer, STATUS := integer_variable)	Create a message port able to hold up to *integer* (default 4) messages, and return the PORT value in port_variable.
CREATE_PROCESS(process_variable, process_block_name,argument-list, EXIT := integer_variable, STATUS := integer_variable)	Create a subprocess running *process_block_name*, with optional arguments supplied to the subprocess, and return the PROCESS value in *process_variable*. Optional integer variables receive the final (exit) status of the subprocess and the procedure's completion status.
CREATE_SEMAPHORE(sem_variable, initial_count, maximum_count, STATUS := integer_variable)	Create a semaphore with the specified *initial_count* and *maximum_count*, and return the SEMAPHORE value in *sem_variable*.

Table C-1. Continued

Call Format	Meaning
CURRENT_PROCESS(process_variable, STATUS := integer_variable)	Return the value of the current process in *process_variable.*
ELN$DEALLOCATE_STACK(stack_size, *status*)	Trim the stack by *stack_size.* The optional INTEGER variable *status* receives the completion status. Include module $STACK_UTILITY.
ELN$DECLARE_EXIT_HANDLER(exit_handler, *exit_context*)	Declare an exit handler, identified by *exit_handler* and *exit_context,* to be called upon termination of a job with the EXIT procedure. Include module $EXIT_UTILITY.
DELETE(system_value, STATUS := integer_variable)	Delete the AREA, DEVICE, EVENT, MESSAGE, NAME, PORT, PROCESS, or SEMAPHORE value from the system.
ELN$DELETE_FILE(file_name, *status*, *resultant_file_name*)	Delete the specified file (up to 255 characters in the file specification) and return the status in the optional INTEGER variable *status.* The optional *resultant_file_name* returns the resultant filename string of the deleted file. Include module $FILE_UTILITY.
ELN$DELETE_MUTEX(mutex_variable, *status*)	Delete the semaphore associated with *mutex_variable* and return the status in the optional INTEGER variable *status.* Include module $MUTEX.
ELN$DIRECTORY_CLOSE(dir_file, *status*)	Close *dir_file* (type ELN$DIR_FILE) and return the completion status in the optional INTEGER variable *status.* Include module $FILE_UTILITY.
ELN$DIRECTORY_LIST(dir_file, directory_name, file_name, *status*, *file_attributes*)	Search *dir_file* (type ELN$DIR_FILE) for the next file specification and return it in *file_name* (type VARYING_STRING(255) variable), with completion status in the optional INTEGER variable *status.* The variable *directory_name* (type VARYING_STRING(255)) receives the resultant directory specification if more than one directory is traversed. The optional *file_attributes* (type FILE$ATTRIBUTES_RECORD) supplies a pointer to the file attributes record. Include module $FILE_UTILITY.

Table C-1. Continued

Call Format	Meaning
ELN$DIRECTORY_OPEN(dir_file, search_name, volume_name, directory_name, *status*, server_name, *file_attributes*)	Open *dir_file* (type ELN$DIR_FILE) in preparation for DIRECTORY_LIST, searching for the specified directory (*search_name*, up to 255 characters). If the search is successful, return with completion status in the optional INTEGER variable *status* and the volume name and directory name in the VARYING_STRING(255) variables *volume_name* and *directory_name*. The optional *server_name* receives the resultant node specification or server process port name. The optional *file_attributes* (type FILE$ATTRIBUTES_RECORD) supplies a pointer to the file attributes record. Include module $FILE_UTILITY.
DISABLE_ASYNCH_EXCEPTION(STATUS := integer_variable)	Disable delivery of asynchronous exceptions to the calling process.
DISABLE_INTERRUPT(priority)	Disable interrupts with interrupt priority level less than or equal to *priority*.
DISABLE_SWITCH(STATUS := integer_variable)	Disable process switching for the job from which the procedure is called.
DISCONNECT_CIRCUIT(port, STATUS := integer_variable)	Break circuit, where *port* is the one in the current job.
ELN$DISMOUNT_TAPE_VOLUME(device, *unload*, *status*)	Dismount a File Service tape on the tape drive named *device* (string of up to 30 characters), and return the completion status in the optional INTEGER variable *status*. The optional *unload* is a BOOLEAN expression that specifies whether the tape is unloaded. Include module $TAPE_UTILITY.
ELN$DISMOUNT_VOLUME(device, *status*)	Dismount the volume on the disk drive named *device* (string of up to 30 characters), and return the completion status in the optional INTEGER variable *status*. Include module $DISK_UTILITY.
DISPOSE(pointer, *tag-list*)	Dispose of storage identified by *pointer*; the optional *tag-list* designates a particular variant if *pointer* identifies a variant record type.

Table C-1. Continued

Call Format	Meaning
ELN$DLV_INITIALIZE(device_name, identifier, *maximum_length*, *string_mode*, *use_polling*)	Ready *device_name* DLV device for input and/or output and create all needed data structures. The variable *identifier* receives an identifier to be used to identify the device in subsequent calls. The optional *maximum_length* supplies the maximum string or block length that will be read or written. The optional arguments *string_mode* and *use_polling* are BOOLEAN expressions. Include module $DLV_UTILITY.
ELN$DLV_READ_BLOCK(identifier, block, *timeout)*	Read from serial line device *identifier* until the specified number of characters is read or, optionally, until *timeout* is reached. The characters read are returned in *block*. Include module $DLV_UTILITY.
ELN$DLV_READ_STRING(identifier, strng*)*	Read from serial line device *identifier* until a carriage return character is encountered. The character string read is returned in *strng*. Include module $DLV_UTILITY.
ELN$DLV_WRITE_STRING(identifier, strng*)*	Write the specified character string *strng* to serial line device *identifier*. Include module $DLV_UTILITY.
ELN$DRV_INITIALIZE(device_name, identifier, buffer, buffer_size, output_ports, use_polling)	Ready *device_name* DRV device for input and/or output and create all needed data structures. The variable *identifier* receives an identifier to be used to identify the device in subsequent calls. The variable *buffer* receives a pointer to the I/O buffer of size *buffer_size*. The set of port numbers to be used for output instead of input is specified by *output_ports*, and *use_polling* is a BOOLEAN expression indicating whether the read procedures will poll the device register or use interrupts. Include module $DRV_UTILITY.
ELN$DRV_READ(identifier, prt, word_count*)*	Read *word_count* data words from parallel port *prt* of device *identifier*. The resulting data is stored in the I/O buffer pointed to by the *buffer* parameter returned by DRV_INITIALIZE. Include module $DRV_UTILITY.
ELN$DRV_WRITE(identifier, prt, word_count*)*	Write *word_count* data words (stored in the I/O buffer pointed to by the *buffer* parameter returned by DRV_INITIALIZE) to parallel port *prt* of device *identifier*. Include module $DRV_UTILITY.
ENABLE_ASYNCH_EXCEPTION(STATUS := integer_variable)	Allow delivery of asynchronous exceptions to the calling process.
ENABLE_INTERRUPT	Reenable interrupts.

Table C-1. Continued

Call Format	Meaning
ENABLE_SWITCH(STATUS := integer_variable)	Resume process switching for the calling job.
KER$ENTER_KERNEL_CONTEXT(*status*, target_routine, argument_block)	Call *target_routine* in kernel mode with *argument_block*, the address of the VAX argument list to be passed to the called routine. The optional INTEGER variable *status* receives the completion status of *target_routine*. Include module $KERNEL.
EOF(*file*)	Return TRUE if *file* (default is INPUT) is at end-of-file.
EOLN(*textfile*)	Return TRUE if current component of *textfile* (default is INPUT) is end-of-line.
ESTABLISH(function_name)	Establish EXCEPTION_HANDLER *function_name* for this block.
EXIT(EXIT_STATUS := integer, STATUS := integer_variable)	End current process, with optional *integer* exit status delivered to creator.
EXP(expression)	Return the value *eexpression*.
FIND(file,record_number)	Position direct-access *file* at indicated record for input (see also LOCATE).
FIND_FIRST_BIT_CLEAR(vector,*start_index*)	Return position of first zero (FALSE) bit in *vector*, a PACKED ARRAY OF BOOLEAN, starting at given *start_index* or first bit.
FIND_FIRST_BIT_SET(vector,*start_index*)	Return position of first one (TRUE) bit in *vector*, a PACKED ARRAY OF BOOLEAN, starting at given *start_index* or first bit.
FIND_MEMBER(string,charset)	Return position in *string* of first character that is a member of set *charset*.
FIND_NONMEMBER(string,charset)	Return position in *string* of first character that is *not* a member of set *charset*.
FLUSH(file)	Flush I/O buffers associated with *file*.
KER$FREE_MAP(*status*,count, number, dev_object)	Free *count* UNIBUS or QBUS map registers, starting with register *number*, previously allocated by ALLOCATE_MAP for device *dev_object*. The optional *status* is an INTEGER variable. Include module $KERNEL.
FREE_MEMORY(size,virtual_address, STATUS := integer_variable)	Free *size* bytes of memory at *virtual_address*, previously allocated by ALLOCATE_MEMORY.
KER$FREE_PATH(*status*, number, dev)	Free UNIBUS datapath *number*, previously allocated for DEVICE *dev* by ALLOCATE_PATH. The optional *status* is an INTEGER variable. Include module $KERNEL.

Table C-1. Continued

Call Format	Meaning
GET(file)	Advance file to next component and move its contents into *file* ↑.
GET_CONTROL_KEY(textfile, keys)	Wait for control key identified by set *keys* (SET OF 0..31) to be pressed at the terminal to which *textfile* is opened.
ELN$GET_STATUS_TEXT(msgid, flags, result_string)	Get text associated with the message identified by the integer *msgid*, format the text as specified by *flags*, and return the resulting string in the VARYING_STRING(255) variable *result_string*. The argument *flags* is a set expression with possible elements STATUS$SEVERITY, STATUS$TEXT, STATUS$IDENT, and STATUS$FACILITY; the presence of an element means that the corresponding field is included in the result string; an empty set means that all fields are included. Include module $GET_MESSAGE_TEXT and link the program with the desired message object modules.
GET_TIME(large_integer_variable, STATUS := integer_variable)	Return current system time in *large_integer_variable*.
KER$GET_USER(*status*, *circuit*, *username*, *uic*)	Return *username* and/or *uic* of either the calling process or the partner process connected by a circuit to the caller's port; the PORT value of the partner process's port is supplied by *circuit*. The optional INTEGER variable *status* receives the completion status. Include module $KERNEL.
HEX(expression,*length*,*digits*)	Return string giving hexadecimal notation for *expression*, with optionally specified *length* and minimum number of significant *digits*.
INDEX(string,substring)	Return position of *substring* in *string*.
ELN$INIT_TAPE_VOLUME(device, volume, *density*, *status*)	Initialize a File Service tape for use as a file-structured volume on the tape drive named *device* (string of up to 30 characters), giving it the volume label *volume* (string of up to 6 characters). The optional *density* is an INTEGER value that supplies the density (in bytes per inch) that the tape will be initialized to. The optional *status* is an INTEGER variable. Include module $TAPE_UTILITY.

Table C-1. Continued

Call Format	Meaning
ELN$INIT_VOLUME(device, volume, *default_extension*, *username*, *owner*, *volume_protection*, *file_protection*, *record_protection*, *accessed_directories*, *maximum_files*, *user_directories*, *file_headers*, *windows*, *cluster_size*, *index_position*, *data_check*, *share*, *group*, *system*, *verified*, bad_list, *status*)	Initialize the disk volume on the drive named *device* (string of up to 30 characters), giving it the volume name *volume* (string of up to 12 characters), and using the bad block list supplied by *bad_list* (a variable-length array of the flexible type DSK$_BADLIST). The optional *status* is an INTEGER variable. Include module $DISK_UTILITY.
INITIALIZATION_DONE(STATUS := integer_variable)	Inform the kernel that the current process has completed its initialization sequence.
ELN$INITIALIZE_AREA_MUTEX(mutex, area, *status*)	Initialize a new mutual exclusion semaphore that uses an AREA object as the synchronization object and return the status in the optional INTEGER variable *status*. The MUTEX variable *mutex* receives the new MUTEX value, and the AREA variable *area* receives the identifier of the new area. Include module $MUTEX.
INSERT_ENTRY(header_queue_entry, queue_entry, boolean_variable, position)	Insert *queue_entry* in queue identified by *header_queue_entry* at *position* QUEUE$HEAD or QUEUE$TAIL; the *boolean_variable* receives TRUE if the inserted entry was the first.
INVOKE(pointer, routine_type, *argument-list*)	Invoke a routine of the given *routine_type* (procedure or function type name), where *pointer* is the address of the routine obtained with ADDRESS and *argument-list* is a valid argument list for the routine.
JOB_PORT(port_variable, STATUS := integer_variable)	Return PORT value identifying the caller's job port in *port_variable*.

Table C-1. Continued

Call Format	Meaning
ELN$KWV_INITIALIZE(device_name, identifier, mode, clock_rate, *maximum_values*, *re_initialize*, *use_polling*, *status*)	Ready *device_name* KWV device for input and create all needed data structures. The variable *identifier* receives a longword identifier to be used to identify the device in subsequent calls. The mode in which the device is to be operated is determined by *mode*, and *clock_rate* supplies the clock frequency to be used. The maximum number of data values that can be read from the device in a single call to KWV_READ is optionally supplied by *maximum_values*. The optional arguments *re_initialize* and *use_polling* are BOOLEAN expressions. The optional INTEGER variable *status* receives the completion status. Include module $KWV_UTILITY.
ELN$KWV_READ(identifier, value_count, st2_go_enable, data_array_ptr, *status*)	Read *value_count* time values from KWV device *identifier* and store them in a data array. The variable *data_array_ptr* receives the address of the array containing data from the device; *st2_go_enable* is a BOOLEAN expression. The optional INTEGER variable *status* receives the completion status. Include module $KWV_UTILITY.
ELN$KWV_WRITE(identifier, st2_go_enable, *tick_count*, *status*)	Set up the KWV device *identifier* to generate the clock-overflow signal. The optional *tick_count* supplies an interval in clock ticks after which a clock-overflow signal is asserted; *st2_go_enable* is a BOOLEAN expression. The optional INTEGER variable *status* receives the completion status. Include module $KWV_UTILITY.
LENGTH(string)	Return length of *string*; for varying strings, this is the current length.
LN(expression)	Return natural (base *e*) logarithm of *expression*.
ELN$LOAD_PROGRAM(file_name, program_name, kernel_mode, start_with_debug, power_recovery, kernel_stack_size, initial_user_stack_size, message_limit, job_priority, process_priority, *status*)	Load *file_name* into a currently running VAXELN system, after which CREATE_JOB will start the program running; *program_name* supplies the name by which the program will be known for the CREATE_JOB call. The starting job and process priorities are specified by *job_priority* and *process_priority*, respectively. The *kernel_mode*, *start_with_debug*, and *power_recovery* arguments are BOOLEAN expressions; *kernel_stack_size*, *initial_user_stack_size*, and *message_limit* are INTEGER values. The optional INTEGER variable *status* receives the completion status. Include module $LOADER_UTILITY.

Table C-1. Continued

Call Format	Meaning
ELN$LOAD_UNIBUS_MAP(map_register, buffer, buffer_size, *spt_address*, *datapath*)	Load UNIBUS or QBUS map registers for use by a direct memory access UNIBUS or QBUS device, where *map_register* is a pointer to the first register allocated by ALLOCATE_MAP, *buffer* (typeBYTE_DATA(buffer_size)) is a variable representing the I/O buffer, *buffer_size* is the integer buffer size, and *spt_address* and *datapath* are the optional pointers to the system page table and the datapath number, respectively. Include module $UNIBUS.
LOCATE(file,record_number)	Position direct-access *file* at indicated *record_number* for output.
ELN$LOCK_MUTEX(mutex)	Lock the MUTEX value *mutex*. Include module $MUTEX.
KER$MEMORY_SIZE(*status*, memory_size, free_size, largest_size)	Scan the kernel memory database and return, in 512-byte pages, the initial *memory_size*, the current *free_size*, and the *largest_size* contiguous block of free memory. The optional *status* is an INTEGER variable. Include module $KERNEL.
MFPR(register_number)	Return integer contents of processor register identified by *register_number*.
ELN$MOUNT_TAPE_VOLUME(device, *volume*, *block_size*, *status*)	Mount a File Service tape on the tape drive named *device* (string of up to 30 characters), using the optional volume label *volume* (string of up to 6 characters). The optional *block_size* supplies an INTEGER value that determines the number of bytes in each block of a newly created file. The optional *status* is an INTEGER variable. Include module $TAPE_UTILITY.
ELN$MOUNT_VOLUME(device, *volume*, *status*)	Mount the disk volume in the drive named *device* (string of up to 30 characters), using the optional volume name *volume* (string of up to 12 characters); if *volume* is omitted, the procedure mounts whatever disk is loaded in the drive. The optional *status* is an INTEGER variable. Include module $DISK_UTILITY.
MOVE_PSL	Return current contents of VAX processor status longword as 32-bit integer.
MTPR(register_number, expression)	Write *expression* (integer, pointer, or LONG data item) to processor register identified by *register_number*.
NEW(pointer, *tag-list*)	Allocate storage for variable of *pointer*'s associated type and validate *pointer* to identify it; the optional *tag-list* designates a particular variant if *pointer* identifies a variant record type.

Table C-1. Continued

Call Format	Meaning
OCT(expression,*length*,*digits*)	Return string giving octal notation for *expression*, with optionally specified *length* and minimum number of significant *digits*.
ODD(integer)	Return TRUE if *integer* is odd.
OPEN(file, FILE_NAME:= specification, HISTORY := file_history, RECORD_LENGTH := length RECORD_LOCKING := lock, ACCESS_METHOD := access, RECORD_TYPE := record_type, CARRIAGE_CONTROL := control, DISPOSITION := disposition, SHARING := sharing, CIRCUIT := action, APPEND := append, BUFFERING := buffering, BUFFERSIZE := buffer_size, CONTIGUOUS := contiguous, EXTENDSIZE := extend_size, FILESIZE := file_size, TRUNCATE := truncate, FILE_ATTRIBUTES := file_attributes, OWNER := owner_uic, PROTECTION := protection_value, STATUS := integer_variable)	Open *file* for I/O, with optional file *specification*, *file_history* (HISTORY$OLD [default], —NEW, —UNKNOWN, or —READONLY), maximum record *length* for textfile (1–32,767, default 133 for new files), record locking (for shareable files) either TRUE or FALSE, *access* method (ACCESS$DIRECT, default —SEQUENTIAL), *record_type* (RECORD$FIXED or —VARIABLE [default for new textfiles]), carriage *control* (CARRIAGE$LIST [textfile], —NONE, —FORTRAN), *disposition* on closing (DISPOSITION$SAVE [default], —DELETE), *sharing* while open (SHARE$NONE, —READONLY, —READWRITE), and circuit *action*, if the "file" variable is used for circuit transmissions (CIRCUIT$CONNECT creates an unnamed port and connects it to a named port; CIRCUIT$ACCEPT creates a named port (where the name is the given file name), and waits for a connection request on that port. If *action* is specified, omit all other arguments except *file* and *specification*.) If *append* is TRUE, the file is initially positioned at end-of-file. If *contiguous* is TRUE, the file is allocated contiguously. If *truncate* is TRUE, the file is truncated to its minimum size. The optional *_size*s are INTEGER expressions. If *buffering* is TRUE (default), file I/O is buffered. The optional *file_attributes* argument (type FILE$ATTRIBUTES_RECORD) supplies a pointer to the file attributes record; include module $FILE_UTILITY to use this argument. The optional *owner_uic* specifies the user identification code of the owner of the file, and the optional *protection_value* supplies a protection code of type FILE$PROTECTION for the file.
ORD(ordinal_expression)	Return ordinal number of *ordinal_expression* result.
PACK(unpacked_array, ordinal_index, packed_array)	Copy elements from *unpacked_array* to *packed_array*, beginning with element at *unpacked_array[ordinal_index]*.
PAGE(*textfile*)	Write subsequent output to *textfile* (default OUTPUT) on new page.

Table C-1. Continued

Call Format	Meaning
PHYSICAL_ADDRESS(pointer)	Return integer denoting the physical address of the variable identified by *pointer*. Include module $PHYSICAL_ADDRESS.
PRED(ordinal_expression)	Return value of *ordinal_expression*'s type with next lower ordinal number.
PRESENT(optional_parameter_name)	Return TRUE if argument was supplied for the named optional parameter.
PROBE_READ(variable_reference)	Return TRUE if first and last bytes of *variable_reference* are accessible for reading in current processor mode.
PROBE_WRITE(variable_reference)	Return TRUE if first and last bytes of *variable_reference* are accessible for writing in current processor mode.
PROGRAM_ARGUMENT(position)	Return program argument at *position* in argument list.
PROGRAM_ARGUMENT_COUNT	Return number of arguments submitted to calling program.
ELN$PROTECT_FILE(file_name, *owner*, *protection*, *status*, *resultant_file_name*)	Change the file *owner* and/or the *protection* code for *file_name* and return the status in the optional INTEGER variable *status*. The optional *resultant_file_name* returns the resultant filename of the file. Include module $FILE_UTILITY.
PUT(file)	Append contents of buffer variable (file ↑) to *file*.
KER$RAISE_DEBUG_EXCEPTION(*status*, job_id, process_id)	Raise the asynchronous exception KER$_DEBUG_SIGNAL in the specified context. The optional *status* is an INTEGER variable. Include module $KERNEL.
RAISE_EXCEPTION(name, argument-list, STATUS := integer_variable)	Raise exception *name* in the calling process, with additional arguments, if any, given by *argument-list*. All arguments are integers.
KER$RAISE_PROCESS_EXCEPTION(*status*, process_var)	Raise the asynchronous exception KER$_PROCESS_ATTENTION in the specified process. The optional *status* is an INTEGER variable. Include module $KERNEL.
READ(*file*, target-list)	Read values from input *file* (default INPUT) and assign to variables in *target-list*.
READLN(*textfile*, *target-list*)	Read values from current line of input *textfile* (default INPUT) and assign to optional variables in *target-list*.

Table C-1. Continued

Call Format	Meaning
READ_REGISTER(device_register)	Return contents of register reprepesented by *device_register* (INTEGER, pointer, or PACKED record variable).
RECEIVE(message_variable, pointer_variable, port, SIZE := integer_variable, DESTINATION := port_variable, REPLY := port_variable, STATUS := integer_variable)	Receive message from *port*, return its MESSAGE value in *message_variable* and a pointer to its data part in *pointer_variable*; INTEGER variables optionally receive the data area's size in bytes and the procedure's completion status; PORT variables optionally receive the destination port specified by the sender and a port for replies.
REMOVE_ENTRY(header_queue_entry, queue_entry_pointer_variable, boolean_variable, position)	Remove queue entry from queue identified by *header_queue_entry* at *position* QUEUE$HEAD, —TAIL, or —CURRENT; *queue_entry_pointer_variable* receives a pointer (↑ QUEUE_ENTRY) to the removed entry; the *boolean_variable* receives TRUE if the queue is empty after the removal.
ELN$RENAME_FILE(old_filename, new_filename, *status*, *resultant_old_filename*, *resultant_new_filename*)	Rename *old_filename* to *new_filename* and return the completion status in the optional INTEGER variable *status*. The optional *resultant_old_filename* and *resultant_new_filename* return the resultant filename strings. Include module $FILE_UTILITY.
RESET(file)	Reset *file* to inspection mode and first record (if any), in preparation for input.
RESUME(process, STATUS := integer_variable)	Resume a previously suspended *process*.
REVERT	Cancel exception handling in current block.
REWRITE(file)	Erase records in *file* in preparation for output.
ROUND(floating_expression)	Return *floating_expression* rounded to nearest integer.
SEND(message, port, SIZE := size, REPLY := port, EXPEDITE := boolean_expression STATUS := integer_variable)	Send *message* to *port*, optionally specifying the data area's *size* in bytes and a *port* for replies. Optionally, expedite the message.
SET_JOB_PRIORITY(integer, STATUS := integer_variable)	Set priority of current job to *integer* (0–31, 0 highest).

Table C-1. Continued

Call Format	Meaning
SET_PROCESS_PRIORITY(process, integer, STATUS := integer_variable)	Set priority of *process* to *integer* (0–15, 0 highest).
KER$SET_PROTECTION(*status*, size, base_address, code)	Set protection of *size* bytes of memory at virtual *base_address* to *code* (0 for read-only access, 1 for read/write access, 2 for no access). The optional INTEGER variable *status* receives the completion status. Include module $KERNEL.
SET_TIME(nonnegative_large_integer, STATUS := integer_variable)	Set current time to *nonnegative_large_integer*.
KER$SET_USER(*status*, username, uic)	Set *username* and *uic* of the current process. The optional INTEGER variable *status* receives the completion status. Include module $KERNEL.
SIGNAL(value, STATUS := integer_variable)	Signal *value* of type AREA, EVENT, SEMAPHORE, or PROCESS.
SIGNAL_DEVICE(DEVICE_NUMBER := integer, STATUS := integer_variable)	Signal DEVICE object from interrupt service routine; the *integer* optionally supplies the index of the value to signal in an ARRAY[0..*n*] OF DEVICE, where $n \leq 15$.
SIN(expression)	Return sine of integer, REAL, or DOUBLE *expression* (angle in radians).
SIZE(item,*tag-list*)	Return size in bytes of *item*, an addressable variable or named type; the optional *tag-list* designates a particular variant if *item* is a variant record.
SQR(expression)	Return square of INTEGER, REAL, or DOUBLE *expression*.
SQRT(expression)	Return square root of nonnegative INTEGER, REAL, or DOUBLE *expression*.
START_QUEUE(queue_entry_variable)	Initialize *queue_entry_variable* for use as a queue header.
SUBSTR(string, position, *length*)	Refer to substring of *string* at *position* (first character is 1); *length* optionally specifies the substring's length (default is the rest of *string*).
SUCC(ordinal_expression)	Return value of *ordinal_expression*'s type with next higher ordinal number.
SUSPEND(process, STATUS := integer_variable)	Suspend the execution of *process*.

Table C-1. Continued

Call Format	Meaning
TIME_FIELDS(large_integer)	Return TIME_RECORD data item representing fields of *large_integer* time. The fields in type TIME_RECORD are unsigned words (16 bits) named *year*, *month*, *day*, *hour*, *minute*, *second*, and *hundredth*. If *large_integer* is a time interval (negative), the returned *year* and *month* are 0, and *day* is the number of days in the time interval, which must be less than 10 thousand.
TIME_STRING(large_integer)	Return string representing time or time interval given by *large_integer*.
TIME_VALUE(string)	Return LARGE_INTEGER value representing time or integer described by *string*; time format: 'dd-mmm-yyyy□hh:mm:ss.cc'; interval format: 'dddd□hh:mm:ss.cc', where □ is a space.
TOTAL_ARGUMENT_COUNT	Return number of arguments in VAX argument list passed to current routine.
TRANSLATE_NAME(port_variable, string, scope, STATUS := integer_variable)	Translate *string*, searching in *scope* NAME$LOCAL, NAME$UNIVERSAL, or NAME$BOTH, and return the associated PORT value in *port_variable*.
TRANSLATE_STRING(original_string, translation_string, OLDCHARS := old_string)	Replace occurrences of *old_string* characters in *original_string* with corresponding *translation_string* characters, and return the resulting string. The result of TRANSLATE_STRING(string,'AEIOU',OLDCHARS:='aeiou') is *string* with all lowercase vowels translated to uppercase. The default for *old_string* is the character set in ascending order.
TRUNC(floating_expression)	Return integer formed by truncating REAL or DOUBLE *floating_expression*.
ELN$UNIBUS_MAP(dev, buffer, buffer_size, unibus_address)	Map memory buffers for direct memory access by UNIBUS or QBUS device *dev*, with BYTE_DATA *buffer* of size *buffer_size*, and return *unibus_address* of first register. Include module $UNIBUS.
ELN$UNIBUS_UNMAP(dev,buffer, buffer_size, unibus_address)	Unmap memory buffers previously mapped for direct memory access by UNIBUS or QBUS device *dev*, with BYTE_DATA *buffer* of size *buffer_size*; *unibus_address* is the address of the first register. Include module $UNIBUS.
ELN$UNLOAD_PROGRAM(program_name, *status*)	Unload program_*name* from a currently running VAXELN system. The optional INTEGER variable *status* receives the completion status. Include module $LOADER_UTILITY.
ELN$UNLOCK_MUTEX(mutex)	Unlock MUTEX value *mutex*. Include module $MUTEX.

Table C-1. Continued

Call Format	Meaning
UNPACK(packed_array, unpacked_array, ordinal_index)	Copy all elements from *packed_array* to *unpacked_array*, with first assignment to *unpacked_array[ordinal_index]*.
KER$UNWIND(*status*, new_fp, new_pc)	Unwind call stack to new location. The optional INTEGER variable *status* receives the completion status. Include module $KERNEL.
WAIT_ALL(object-list, RESULT := integer_variable, TIME := large_integer, STATUS := integer_variable)	Make calling process wait for all AREA, DEVICE, EVENT, PORT, PROCESS, or SEMAPHORE values in *object-list* to satisfy the wait. Zero to four object values can be specified; the optional *integer_variable* receives a nonzero value if the system values satisfied the wait or 0 if the procedure timed out. The optional *large_integer* specifies a time interval or absolute time defining the timeout; the timeout is irrelevant, and the wait result is nonzero, if the necessary conditions were satisfied before the call.
WAIT_ANY(object-list, RESULT := integer_variable, TIME := large_integer, STATUS := integer_variable)	Make calling process wait for any object in *object-list* to satisfy the wait. If one or more object value is specified, the optional *integer_variable* receives the position of the argument satisfying the wait or 0 if the procedure timed out. The optional *large_integer* specifies a timeout as for WAIT_ALL.
WRITE(*file*, source-list)	Write expressions in *source-list* to *file* (default OUTPUT); if *file* is a textfile, the source expressions are converted to character-string representations; otherwise, the types of the source expressions must be the same as the type of *file* ↑. For textfile output, source expressions can be suffixed with field specifiers of the form "width:fraction". For integers and floating-point numbers, any specified field width is the **minimum** width.
WRITELN(*textfile, source-list*)	Equivalent to WRITE(textfile,source-list);WRITELN(textfile), where the WRITELN appends the end-of-line component. The default *textfile* is OUTPUT; if no *source-list* is supplied, any current, partial line is ended and the next output begins on the next line.

Table C-1. Continued

Call Format	Meaning
WRITE_REGISTER(register, *expression*)	Write compatible expression to *register*, which must be an INTEGER variable, a pointer with the BYTE, WORD, or LONG attribute, or a PACKED record with one of these attributes. If *expression* is omitted, the register is zeroed.
WRITE_REGISTER(register, *assignment-list*)	Write compatible expressions to *register*, which must be a PACKED record with the BYTE, WORD, or LONG attribute. Each *assignment* has the form "fieldname : = expression", where the field names are those of the record. Any record field without an assignment is zeroed; if no assignments are given, the entire register is zeroed. Any bits in the register not represented by field names are cleared always.
XOR(boolean_value,boolean_value)	Return TRUE if the two arguments have different TRUE/FALSE values, otherwise FALSE.
XOR(seta,setb)	Return a set whose elements are present in either set argument but not in both.
ZERO	Return binary zero in the target of the assignment (for example, A : = ZERO or VAR a : ARRAY[1..5,1..10] OF INTEGER : = (REPEAT ZERO).

Index

A

B

C

D

E

F

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z

VAXELN PASCAL
Language Reference Manual
AA-EU39A-TE

READER'S COMMENTS

Your comments and suggestions will help us in our continuous effort to improve the quality and usefulness of our handbooks.

What is your general reaction to this handbook? (format, accuracy, completeness, organization, etc.) _____

_____

_____

_____

What features are most useful? _____

_____

_____

_____

Does the publication satisfy your needs? _____

_____

_____

What errors have you found? _____

_____

_____

_____

Additional comments _____

_____

_____

Name _____

Title _____

Company _____ Dept. _____

Address _____

City _____ State _____ Zip _____

(staple here)

--------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

Attention Publications Manager
Digital Equipment Corporation
2265 116 Avenue Northeast
Bellevue,
Washington, 98004

--------Do Not Tear - Fold Here and Tape ----------