VMS

Introduction to VMS System Routines

Order Number: AA–LA66B–TE

Introduction to VMS System Routines

Order Number: AA–LA66B–TE

June 1990

This manual describes the documentation format for the system routines, the VAX Procedure Calling and Condition Handling Standard, and the VAX language implementation tables.

Revision/Update Information:	This manual supersedes the *Introduction to VMS System Routines*, Version 5.0.
Software Version:	VMS Version 5.4

digital equipment corporation
maynard, massachusetts

June 1990

The postpaid Reader's Comments forms at the end of this document request your critical evaluation to assist in preparing future documentation.

ZK4512

Production Note

This book was produced with the VAX DOCUMENT electronic publishing system, a software tool developed and sold by Digital. In this system, writers use an ASCII text editor to create source files containing text and English-like code; this code labels the structural elements of the document, such as chapters, paragraphs, and tables. The VAX DOCUMENT software, which runs on the VMS operating system, interprets the code to format the text, generate a table of contents and index, and paginate the entire document. Writers can print the document on the terminal or line printer, or they can use Digital-supported devices, such as the LN03 laser printer and PostScript printers (PrintServer 40 or LN03R ScriptPrinter), to produce a typeset-quality copy containing integrated graphics.

Contents

Preface

Intended Audience

This manual is intended for all programmers who call VMS-supplied system routines.

Document Structure

This manual contains two chapters and an appendix.

- Chapter 1 describes the format used to document system routines.
- Chapter 2 describes the VAX Procedure Calling and Condition Handling Standard. This standard explains programming mechanisms that are used with the VAX hardware procedure-calling mechanism.
- Appendix A describes VMS data types, the VMS Usage entry, and the VAX language implementation tables.

Associated Documents

The following four manuals document the VMS-supplied system routines:

- *VMS System Services Reference Manual*
- *VMS Run-Time Library Routines Volume*
- *VMS Record Management Services Manual*
- *VMS Utility Routines Manual*

The *VAX Architecture Reference Manual* and the *VAX Architecture Handbook* also contain information about the VAX architecture and its procedure-calling mechanisms.

Conventions

The following conventions are used in this manual:

Convention	Meaning
Ctrl/x	A sequence such as Ctrl/x indicates that you must hold down the key labeled Ctrl while you press another key or a pointing device button.
$\boxed{\text{Return}}$	In examples, a key name is shown enclosed in a box to indicate that you press a key on the keyboard. (In text, a key name is not enclosed in a box.)

Convention	Meaning
. . .	In examples, a horizontal ellipsis indicates one of the following possibilities: • Additional optional arguments in a statement have been omitted. • The preceding item or items can be repeated one or more times. • Additional parameters, values, or other information can be entered.
. . .	A vertical ellipsis indicates the omission of items from a code example or command format; the items are omitted because they are not important to the topic being discussed.
( )	In format descriptions, parentheses indicate that, if you choose more than one option, you must enclose the choices in parentheses.
[ ]	In format descriptions, brackets indicate that whatever is enclosed within the brackets is optional; you can select none, one, or all of the choices. (Brackets are not, however, optional in the syntax of a directory name in a file specification or in the syntax of a substring specification in an assignment statement.)
{}	In format descriptions, braces surround a required choice of options; you must choose one of the options listed.
boldface text	Boldface text represents the introduction of a new term or the name of an argument, an attribute, or a reason.
italic text	Italic text represents information that can vary in system messages (for example, Internal error *number*).
UPPERCASE TEXT	Uppercase letters indicate that you must enter a command (for example, enter OPEN/READ), or they indicate the name of a routine, the name of a file, the name of a file protection code, or the abbreviation for a system privilege.
	Hyphens in coding examples indicate that additional arguments to the request are provided on the line that follows.
numbers	Unless otherwise noted, all numbers in the text are assumed to be decimal. Nondecimal radixes—binary, octal, or hexadecimal—are explicitly indicated.

1 Documentation Format for System Routines

Each system routine is documented using a structured format. This chapter discusses the main categories in this format, the information presented under each, and the format used to present the information.

1.1 Overview

This chapter explains where to find information on routines and how to read that information correctly. Subsequent chapters cover the VAX Procedure Calling and Condition Handling Standard and VMS Data Types.

Note: **The documentation format described in this chapter is generic; portions of it are used or not used, as appropriate, in the four VMS manuals that document system routines.**

VMS System Services Reference Manual
VMS Run-Time Library Routines Volume
VMS Utility Routines Manual
VMS Record Management Services Manual

Some main categories in the routine format contain information requiring no explanation beyond that given in Table 1–1. However, additional information, presented in this manual, is required for the following categories:

- Format
- Returns
- Arguments
- Condition Values Returned

Table 1–1 Main Heading in the Documentation Format for System Routines

Main Heading	Description
Routine Name	Always present. The routine entry-point name appears at the top of the first page. It is usually, though not always, followed by the English text name of the routine.
Routine Overview	Always present. The routine overview appears directly below the routine name; the overview explains, usually in one or two sentences, what the routine does.

(continued on next page)

Table 1–1 (Cont.) Main Heading in the Documentation Format for System Routines

Main Heading	Description
Format	Always present. The format heading follows the routine overview. The format gives the routine entry-point name and the routine argument list.
Returns	Always present. The returns heading follows the routine format. It explains what information is returned by the routine.
Arguments	Always present. The arguments heading follows the returns heading. Detailed information about each argument is provided under the arguments heading. If a routine takes no arguments, it is indicated by the word "None."
Description	Optional. The description heading follows the arguments heading. The description section contains information about specific actions taken by the routine: interaction between routine arguments, if any; operation of the routine within the context of VMS; user privileges needed to call the routine, if any; system resources used by the routine; and user quotas that might affect the operation of the routine. Note that any restrictions on the use of the routine are always discussed first in the description section; for example, any required user privileges or necessary system resources are explained first. For some simple routines, a description section is not necessary because the routine overview provides the needed information.
Condition Values Returned	Always present. The condition values returned section follows the description section. It lists the condition values (typically status or completion codes) that are returned by the routine.
Example	Optional. The examples heading appears following the condition values returned heading. The example section contains one or more programming examples that illustrate how to use the routine, and is followed by an explanation. All examples have been tested and should run when compiled (or assembled) and linked. Incomplete examples and code fragments do not appear under the examples heading. Throughout the manuals that document system routines, examples are provided in as many different programming languages as possible.

1.2 Format Heading

The following three types of information can be present in the format heading:

- Procedure call format

- JSB (Jump to Subroutine) format
- Explanatory text

All system routines have a procedure call format, but not all system routines have JSB formats; most do not. If a routine has a JSB format, it always appears after the routine's procedure call format.

Procedure Call Format

The procedure call format ensures that a routine call conforms to the procedure call mechanism described in the VAX Procedure Calling and Condition Handling Standard in Chapter 2; for example, an entry mask is created, registers are saved, and so on.

Procedure call formats can appear in many forms. The following four examples illustrate the meaning of syntactical elements, such as brackets and commas. General rules of syntax governing how to use procedure call formats are shown in Table 1–2.

Example 1

This example illustrates the standard representation of optional arguments and best describes the use of commas as delimiters. Arguments enclosed within square brackets are optional, but if an optional argument other than a trailing optional argument is omitted, you must include a comma as a delimiter for the omitted argument.

ROUTINE_NAME *arg1[, [arg2][, arg3]]*

Typically, VMS RMS system routines use this format where, at most, three arguments appear in the argument list.

Example 2

When the argument list contains three or more optional arguments, the syntax does not provide enough information. If you omit the optional arguments **arg3** and **arg4** and specify the trailing argument **arg5**, you *must* use commas to delimit the positions of the omitted arguments.

ROUTINE_NAME *arg1, [arg2], nullarg, [arg3], [arg4], arg5*

Typically, VMS system services, utility routines, and VAX Run-Time Library routines contain call formats with more than three arguments.

Example 3

In the following call format example, the trailing four arguments are optional as a group; that is, either you specify **arg2**, **arg3**, **arg4**, and **arg5**, or none of them. Therefore, if you do not specify the optional arguments, you need not use commas to delimit unoccupied positions.

However, if you specify a required argument or a separate optional argument after **arg5**, you must use commas when **arg2**, **arg3**, **arg4**, and **arg5** are omitted.

ROUTINE_NAME *arg1[, arg2, arg3, arg4, arg5]*

Example 4

In the following example, you can specify **arg2** and omit **arg3**. However, whenever you specify **arg3**, you *must* specify **arg2**.

ROUTINE_NAME *arg1[, arg2[, arg3]]*

JSB Call Format

The JSB call format activates the routine code directly, without the overhead of constructing the entry mask or saving registers. You can use the JSB call format only with the VAX MACRO and VAX BLISS languages.

Explanatory Text

Explanatory text might follow the procedure call format or the JSB call format, or both. This text is present only when needed to clarify the format. For example, in the call format, you indicate that arguments are optional by enclosing them in brackets ([ ]). However, brackets alone cannot convey all the important information that might apply to optional arguments. For example, in some routines that have many optional arguments, if you select one optional argument, you must also select another optional argument. In such cases, text following the format clarifies this.

Table 1–2 General Rules of Syntax

Element	Syntax Rule
Entry point names	Entry point names are always shown in uppercase characters.
Argument names	Argument names are always shown in lowercase characters.
Spaces	One or more spaces are used between the entry point name and the first argument, and between each argument.
Braces	Braces surround two or more arguments. You must choose one of the arguments.
Brackets ([ ])	Brackets surround optional arguments. Note that commas too can be optional (see the comma element).
Commas	Between arguments, the comma always follows the space. If the argument is optional, the comma might appear inside the brackets or outside the brackets, depending on the position of the argument in the list and on whether surrounding arguments are optional or required.

(continued on next page)

Table 1–2 (Cont.) General Rules of Syntax

Element	Syntax Rule
Null arguments	A null argument is a place-holding argument. It is used for either of the following reasons: (1) to hold a place in the argument list for an argument that has not yet been implemented by Digital but might be in the future; or (2) to mark the position of an argument that was used in earlier versions of the routine but is not used in the latest version (upward compatibility is thereby ensured because arguments that follow the null argument in the argument list keep their original positions). A null argument is always given the name **nullarg**. In the argument list constructed on the stack when a procedure is called, both null arguments and omitted optional arguments are represented by longword argument list entries containing the value *0*. The programming language syntax required to produce argument list entries containing *0* differ from language to language. See your language user's guide for language-specific syntax.

1.3 Returns Heading

The returns heading contains a description of any information returned by the routine to the caller. A routine can return information to the caller in various ways. The following subsections discuss each possibility and then describe how this returned information is presented.

1.3.1 Condition Values Returned in R0

Most routines return a condition value in register R0. This condition value contains various kinds of information, but most importantly for the caller, it describes (in bits <3:0>) the completion status of the operation. You test the condition value to determine if the routine completed successfully.

If you program in high-level languages, the fact that status information is returned by means of a condition value and that it is returned in a VAX register is of little importance because you receive this status information in the return (or status) variable. The run-time environment established for the high-level language program allows the status information in R0 to be moved automatically to the user's return variable.

Nevertheless, for routines that return a condition value in R0, the returns heading in the documentation contains the following information:

```
VMS Usage:     longword_unsigned
type:          longword (unsigned)
access:        write only
mechanism:     by value
```

The **VMS Usage** entry specifies the VMS data type of the information returned. Because the data type of a condition value in the VMS operating system environment is an unsigned longword, the VMS Usage entry is **longword_unsigned**.

The **type** entry specifies the data type of the information returned. Because the data type of a condition value is an unsigned longword, the type heading is **longword (unsigned)**.

The **access** entry specifies the way in which the called routine accesses the object. Because the called routine is returning the condition value, it is writing into this longword, so the access heading is **write only**.

The **mechanism** heading specifies the passing mechanism used by the called routine in returning the condition value. Because the called routine is writing the condition value directly into R0, the mechanism heading is **by value**. (If the called routine had written the address of the condition value into R0, the passing mechanism would have been **by reference**.)

Note that if a routine returns a condition value in R0, another main heading in the documentation format (Condition Values Returned) describes the possible condition values that the routine can return.

1.3.2 Data in Registers R0 Through R11

Some routines return actual data in the VAX registers. The number of registers needed to contain the data depends on the length (or data type) of the information being returned. For example, a Run-Time Library mathematics routine that is returning the cosine of an angle as a G_ floating point number would use registers R0 and R1 because the length of a G_floating point number is two longwords.

If a routine returns actual data in one or more of the registers R0 through R11, the returns heading in the documentation of that routine contains the following information:

```
VMS Usage:    floating_point
type:         G_floating
access:       write only
mechanism:    by value
```

For example, for the mathematics routine just discussed, the VMS data type is floating_point and the VAX standard data type is G_floating point. The meaning of the contents of the access and mechanism headings are discussed in Sections 1.4.3 and 1.4.4.

In addition, under the returns heading, some text can be provided after the information about the type, access, and mechanism. This text explains other relevant information about what the routine is returning.

For example, because the routine is returning actual data in the VAX registers, the registers cannot be used to convey completion status information. All routines that return actual data in VAX registers must **signal** the condition value, which contains the completion status. Thus, the text under the returns heading points out that the routine signals its completion status.

1.3.3 Condition Values Signaled

Although most routines return condition values in R0, some routines choose to signal their condition values using the VAX Signaling Mechanism. Routines can signal their completion status whether or not they are returning actual data in the VAX registers, but all routines that return actual data in the VAX registers must signal their completion status if they are to return this status information at all.

If a routine signals its completion status, text under the returns heading explains this, and the Condition Values Signaled heading in the documentation format describes the possible condition values that the routine can signal.

Digital's system routines never signal condition values indicating success. Only error condition values are signaled.

1.4 Arguments Heading

Detailed information about each argument is listed in the call format under the arguments heading. Arguments are described in the order in which they appear in the call format. If the routine has no arguments, it is indicated by the word "None."

The following format is used to describe each argument:

```
argument-name
VMS Usage:    VMS data type
type:         argument data type
access:       argument access
mechanism:    argument passing mechanism
```

Next is a paragraph of structured text describing the arguments. Additional information follows, if needed.

1.4.1 VMS Usage Entry

The purpose of the VMS Usage entry is to facilitate the coding of source language data type declarations in application programs. As mentioned previously, argument data types are described in two ways:

- VMS data type
- VAX standard data type

Ordinarily, the VAX standard data type, discussed in Section 1.4.2 would be sufficient to describe the type of data passed by an argument. However, within the VMS operating system environment, many system routines contain arguments whose conceptual nature or complexity, or both, require additional explanation. For instance, when an argument passes the name of an array by reference, the type entry **longword (unsigned)** alone does not indicate that a data structure argument is being referenced. In this particular instance, an accompanying VMS Usage entry, denoting the VMS data type **vector_ longword_unsigned**, further explains that an array of unsigned longwords must be declared.

Note: **The VMS Usage entry is NOT a traditional data type (such as the VAX standard data types byte, word, longword, and so on). It is significant only within the context of the VMS operating system environment and is intended solely to expedite data declarations within application programs.**

Table A–1 in Appendix A lists possible VMS Usage entries and their definitions.

See the appropriate VAX language implementation table (Tables A–2 through A–13) in Appendix A to determine the correct syntax of the type declaration in the language you are using.

1.4.2 Type Entry

In actuality, an argument does not have a data type; rather, the data specified by an argument has a data type. The argument is merely the vehicle for passing data to the called routine. Nevertheless, the phrase **argument data type** is used frequently to describe the data type of the data specified by the argument. This terminology is used because it is more simple and straightforward than the strictly accurate phrase *data type of the data specified by the argument.*

Procedure calls result in the construction of an **argument list** on the stack. (This process is described in Chapter 2.) The argument list is a vector of longwords. The first longword on the list contains a count of the number of remaining longwords, and each remaining longword is one argument. Thus, an **argument** is one longword in the argument list.

Table 1–3 lists each VAX standard data type that can appear for the type entry in an argument description and the VMS-defined symbolic code for each. These symbolic codes are used in descriptors.

For a detailed description of each of the following symbolic codes, see Section 2.8.

Table 1–3 VAX Standard Data Types

Data Type	Symbolic Code
Absolute date and time	DSC$K_DTYPE_ADT
Byte integer (signed)	DSC$K_DTYPE_B
Bound label value	DSC$K_DTYPE_BLV
Bound procedure value	DSC$K_DTYPE_BPV
Byte (unsigned)	DSC$K_DTYPE_BU
COBOL intermediate temporary	DSC$K_DTYPE_CIT
D_floating	DSC$K_DTYPE_D
D_floating complex	DSC$K_DTYPE_DC
Descriptor	DSC$K_DTYPE_DSC

(continued on next page)

Table 1–3 (Cont.) VAX Standard Data Types

Data Type	Symbolic Code
F_floating	DSC$K_DTYPE_F
F_floating complex	DSC$K_DTYPE_FC
G_floating	DSC$K_DTYPE_G
G_floating complex	DSC$K_DTYPE_GC
H_floating	DSC$K_DTYPE_H
H_floating complex	DSC$K_DTYPE_HC
Longword integer (signed)	DSC$K_DTYPE_L
Longword (unsigned)	DSC$K_DTYPE_LU
Numeric string, left separate sign	DSC$K_DTYPE_NL
Numeric string, left overpunched sign	DSC$K_DTYPE_NLO
Numeric string, right separate sign	DSC$K_DTYPE_NR
Numeric string, right overpunched sign	DSC$K_DTYPE_NRO
Numeric string, unsigned	DSC$K_DTYPE_NU
Numeric string, zoned sign	DSC$K_DTYPE_NZ
Octaword integer (signed)	DSC$K_DTYPE_O
Octaword (unsigned)	DSC$K_DTYPE_OU
Packed decimal string	DSC$K_DTYPE_P
Quadword integer (signed)	DSC$K_DTYPE_Q
Quadword (unsigned)	DSC$K_DTYPE_QU
Character string	DSC$K_DTYPE_T
Aligned bit string	DSC$K_DTYPE_V
Varying character string	DSC$K_DTYPE_VT
Unaligned bit string	DSC$K_DTYPE_VU
Word integer (signed)	DSC$K_DTYPE_W
Word (unsigned)	DSC$K_DTYPE_WU
Unspecified	DSC$K_DTYPE_Z
Procedure entry mask	DSC$K_DTYPE_ZEM
Sequence of instruction	DSC$K_DTYPE_ZI

1.4.3 Access Entry

The access entry describes the way in which the called routine accesses the data specified by the argument, or **access method**. The following three methods of access are the most common:

- Read only. Data upon which a routine operates, or data needed by the routine to perform its operation, must be **read** by the called routine. Such data is also called **input** data. When an argument specifies input data, the access entry is read only.

The term **only** is present to indicate that the called routine does not both read and write (that is, **modify**) the input data. Thus, input data supplied by a variable is preserved when the called routine completes execution.

- Write only. Data that the called routine returns to the calling routine must be **written** into a location where the calling routine can access it. Such data is also called **output** data. When an argument specifies output data, the access entry is write only.

  In this context, the term **only** is present to indicate that the called routine does not read the contents of the location either before or after it writes into the location.

- Modify. When an argument specifies data that is both read and written by the called routine, the access entry is modify. In this case, the called routine reads the input data, which it uses in its operation, and then overwrites the input data with the results (the output data) of the operation. Thus, when the called routine completes execution, the input data specified by the argument is lost.

Following is a complete list of access methods that can appear under the access entry in an argument description:

- Read only
- Write only
- Modify
- Function call (before return)
- JMP after unwind
- Call after stack unwind
- Call without stack unwind

For more information, see the VAX Procedure Calling and Condition Handling Standard in Chapter 2 of this manual.

1.4.4 Mechanism Entry

The way in which an argument specifies the actual data to be used by the called routine is defined in terms of the argument **passing mechanism**. There are three basic passing mechanisms:

- By value. When the longword argument in the argument list contains the actual data to be used by the routine, the actual data is said to be passed to the routine by value. In this case, the longword argument contains the actual data; in other words, the argument is the actual data. Because an argument is only one longword in length, only data that can be represented in one longword can be passed by value.

- By reference. When the longword argument in the argument list contains the address of the data to be used by the routine, the data is said to be passed by reference. In this case, the argument is a pointer to the data.

- By descriptor. When the longword argument in the argument list contains the address of a descriptor, the data is said to be passed by descriptor. A descriptor consists of two or more longwords (depending on the type of descriptor used) that describe the location, length, and the VAX standard data type of the data to be used by the called routine. In this case, the argument is a pointer to a descriptor that itself is a pointer to the actual data.

  There are several types of descriptor. Each one contains a value, or **class type**, in the fourth byte of the first longword. The class type identifies the type of descriptor it is. Each class type has a symbolic code.

  Table 1–4 lists the types of descriptors and their corresponding class types. See Section 2.9 for a detailed description of each descriptor class type.

Table 1–4 Descriptor Passing Mechanism Class Types

Passing Mechanism	Descriptor Symbolic Code
By descriptor, fixed-length	DSC$K_CLASS_S
By descriptor, dynamic string	DSC$K_CLASS_D
By descriptor, array	DSC$K_CLASS_A
By descriptor, procedure	DSC$K_CLASS_P
By descriptor, decimal string	DSC$K_CLASS_SD
By descriptor, noncontiguous array	DSC$K_CLASS_NCA
By descriptor, varying string	DSC$K_CLASS_VS
By descriptor, varying string array	DSC$K_CLASS_VSA
By descriptor, unaligned bit string	DSC$K_CLASS_UBS
By descriptor, unaligned bit array	DSC$K_CLASS_UBA
By descriptor, string with bounds	DSC$K_CLASS_SB
By descriptor, unaligned bit string with bounds	DSC$K_CLASS_UBSB

1.4.5 Explanatory Text Entry

For each argument, one or more paragraphs of explanatory text follow the VMS Usage, type, access, and mechanism entries. The first paragraph is highly structured and always contains information in the following sequence:

1 A sentence or a sentence fragment that describes (1) the nature of the data specified by the argument and (2) the way in which the routine uses this data. For example, if an argument were supplying a number, which the routine converts to another data type, the argument description would contain the following sentence fragment:

> Integer to be converted to an F_floating point number

2 A sentence that expresses the relationship between the argument and the data that it specifies. This relationship is the passing mechanism used to pass the data and, for a given argument, is expressed in one of the following ways:

a. If the passing mechanism is by value, the sentence should read as follows:

> The **attrib** argument is a longword that contains (or is) the bit mask specifying the attributes.

b. If the passing mechanism is by reference, the sentence should read as follows:

> The **objtyp** argument is the address of a longword containing a value indicating whether the object is a file or a device.

c. If the passing mechanism is by descriptor, the sentence should read as follows:

> The **devnam** argument is the address of a string descriptor of a logical name denoting a device name.

3 Additional explanatory paragraphs that appear for each argument, as needed. For example, some arguments specify complex data consisting of many discrete fields, each of which has a particular purpose and use. In such cases, additional paragraphs provide detailed descriptions of each such field, symbolic names for the fields, if any, and guidance on their use.

1.5 Condition Values Returned Heading

A condition value is an unsigned longword that has the following uses in the VAX architecture:

- Indicates the success or failure of a called procedure
- Describes an exception condition when an exception is signaled

- Identifies system messages
- Reports program success or failure to the command level

Section 2.5 explains in detail the uses for the longword condition value and what it contains. Figure 2–2 depicts its format.

The documentation heading Condition Values Returned describes the condition values returned by the routine when it completes execution without generating an exception condition. These condition values describe the completion status of the operation.

If a called routine generates an exception condition during execution, the exception condition is **signaled**; the exception condition is then **handled** by a condition handler (either user-supplied or system-supplied). Depending on the nature of the exception condition and on the condition handler that handles the exception condition, the called routine either continues normal execution or terminates abnormally.

If a called routine executes without generating an exception condition, the called routine returns a condition value in one or two of the following four possible ways:

- Condition Values Returned
- Condition Values Returned in the I/O Status Block
- Condition Values Returned in a Mailbox
- Condition Values Signaled

The method used to return the condition value is indicated under the Condition Values Returned heading in the documentation of each routine. These methods are discussed individually in the following subsections.

Under these headings, a 2-column list shows the symbolic code for each condition value the routine can return and an accompanying description. The description explains whether the condition value indicates success or failure and, if failure, what user action might have caused the failure and what you can do to correct it. Condition values that indicate success are listed first.

Symbolic codes for condition values are defined by the system. The symbolic code defined for each condition value equates to a number that is identical to the longword condition value when interpreted as a number. In other words, though the condition value consists of several fields, each of which can be interpreted individually for specific information, the entire longword condition value itself can be interpreted as an unsigned longword integer, and this integer has an equivalent symbolic code.

The three sections that follow discuss the ways in which the called routine returns condition values.

1.5.1 Condition Values Returned

The possible condition values that the called routine can return in general register R0 are listed under the Condition Values Returned heading in the documentation. Most routines return a condition value in this way.

In the documentation of system services that complete asynchronously, both the Condition Values Returned and Condition Values Returned in the I/O Status Block are used. Under the Condition Values Returned heading, the condition values returned by the asynchronous service refer to the success or failure of the system service request—that is, to the status associated with the correctness of the syntax of the call, in contrast to the final status associated with the completion of the service operation. For asynchronous system services, condition values describing the success or failure of the actual service operation—that is, the final completion status—are listed under the Condition Values Returned in the I/O Status Block heading.

1.5.2 Condition Values Returned in the I/O Status Block

The possible condition values that the called routine can return in an I/O status block are listed under the Condition Values Returned in the I/O Status Block heading in the documentation.

The routines that return condition values in the I/O status block are the system services that are completed asynchronously. Each of these asynchronous system services returns to the caller as soon as the service call is queued. This allows the continued use of the calling program during the execution of the service operations. System services that are completed asynchronously all have arguments that specify an I/O status block. When the system service operation is completed, a condition value specifying the completion status of the operation is written in the first word of this I/O status block.

Representing a longword condition value in a word-length field is possible for system services because the high-order word in all system service condition values is *0*. Section 2.5 explains in detail what the fields contain in the longword condition value.

1.5.3 Condition Values Returned in a Mailbox

The possible condition values that the called routine can return in a mailbox are listed under the Condition Values Returned in a Mailbox heading.

Routines such as SYS$SNDOPR that return condition values in a mailbox send information to another process to perform a task. The receiving process performs the action and returns the status of the task to the mailbox of the sending process.

1.5.4 Condition Values Signaled

The possible condition values that the called routine can signal (instead of returning them in R0) are listed under the Condition Values Signaled heading.

Routines that signal condition values as a way of indicating the completion status do so because these routines are returning actual data in one or more of the general registers. Because register R0 is used to convey data, it cannot also receive the condition value.

As mentioned, the signaling of condition values occurs whenever a routine generates an exception condition, regardless of how the routine returns its completion status under normal circumstances.

2 VAX Procedure Calling and Condition Handling Standard

This chapter describes the VAX Procedure Calling and Condition Handling Standard, Version 10.3.

2.1 Introduction

The VAX Procedure Calling Standard is used with the VAX hardware procedure call mechanism. This standard applies to the following:

- All externally callable interfaces in Digital-supported, standard system software
- All intermodule calls to major VAX components
- All external procedure calls generated by standard Digital language processors

This standard does not apply to calls to internal (local) routines or to language support routines. Within a single module, the language processor or programmer can use a variety of other linkage and argument-passing techniques.

The standard defines mechanisms for passing arguments by immediate value, by reference, and by descriptor. However, the immediate value mechanism is intended for use only by VMS system services and within programs written in VAX BLISS, VAX C, or VAX MACRO.

The procedure call mechanism depends on agreement between the calling and called procedures to interpret the argument list. The argument list does not fully describe itself. This standard requires language extensions to permit a calling program to generate some of the argument-passing mechanisms expected by called procedures.

This standard specifies the following attributes of the interfaces between modules:

- Calling sequence—The instructions at the call site and at the entry point
- Argument list—The structure of the list describing the arguments to the called procedure
- Function value return—The form and conventions for the return of the function value as a value or as a condition value to indicate success or failure
- Register usage—Which registers are preserved and who is responsible for preserving them
- Stack usage—Rules governing the use of the stack
- Argument data types—The data types of arguments that can be passed

- Argument descriptor formats—How descriptors are passed for the more complex arguments
- Condition handling—How exception conditions are signaled and how they can be handled in a modular fashion
- Stack unwinding—How the current thread of execution can be aborted efficiently

2.1.1 Goals of the Calling Standard

When the VAX Procedure Calling Standard was developed, the following goals were kept in mind:

- The standard must be applicable to all intermodule callable interfaces in the VAX software system. Specifically, the standard must consider the requirements of VAX MACRO, VAX Ada, VAX BLISS, VAX BASIC, VAX C, VAX COBOL, VAX CORAL, VAX FORTRAN, VAX Pascal, VAX PEARL, VAX PL/I, VAX RPG II, and calls to the operating system and library procedures. The needs of other languages that Digital might support in the future must be met by the standard or by a compatible revision of it.
- The standard should not include capabilities for lower-level components (such as VAX BLISS, VAX MACRO, operating system) that cannot be invoked from the higher-level languages.
- The calling program and called procedure should be writable in different languages. The standard attempts to reduce the need for use of language extensions for mixed-language programs.
- The procedure mechanism must be sufficiently economical in both space and time to be usable as the only calling mechanism within VAX.
- The standard should contribute to the writing of error-free, modular, and maintainable software. Effective sharing and reuse of VAX software modules are significant goals.
- The standard must allow the called procedure to have a variety of techniques for argument handling. Specifically, the called procedure must be able to do the following:
  - Reference arguments indirectly through the argument list
  - Copy atomic data types, strings, and arrays
  - Copy addresses of atomic data types, strings, and arrays
- The standard should provide you with some control over fixing, reporting, and controlling flow on hardware and software exceptions.
- The standard should provide subsystem and application writers with the ability to override system messages to provide a more suitable application-oriented interface.

- The standard should add no space or time overhead to procedure calls and returns that do not establish handlers and should minimize time overhead for establishing handlers at the cost of increased time overhead when exceptions occur.

Some possible attributes of a procedure-calling mechanism were considered and rejected. Specific attributes rejected for the VAX procedure-calling mechanism include the following:

- The procedure mechanism need not provide complete checking of argument data types, data structures, and parameter access. The VAX protection and memory-management system is not dependent upon correct interactions between user-level calling and called procedures. Such extended checking might be desirable in some circumstances, but system integrity is not dependent upon it.
- The VAX procedure mechanism need not provide complete information for an interpretive VMS Debugger. The definition of the debugger includes a debug symbol table (DST) that contains the required descriptive information.

2.1.2 Definitions Used in the VAX Calling Standard

A **procedure** is a closed sequence of instructions that is entered from and returns control to the calling program.

A **function** is a procedure that returns a single value in accordance with the standard conventions for value returning. If additional values are returned, they are returned by means of the argument list.

A **subroutine** is a procedure that returns a known value not in accordance with the standard conventions for value returning. If values are returned, they are returned by means of the argument list.

An **address** is a 32-bit VAX address positioned in a longword item.

An **argument list** is a vector of longwords that represents a procedure parameter list and possibly a function value.

Immediate value is a mechanism for passing input parameters where the actual value is provided in the longword argument list entry by the calling program.

Reference is a mechanism for passing parameters where the address of the parameter is provided in the longword argument list by the calling program.

Descriptor is a mechanism for passing parameters where the address of a descriptor is provided in the longword argument list entry. The descriptor contains the address of the parameter, the data type, size, and additional information needed to describe fully the data passed.

An **exception condition** is a hardware- or software-detected event that alters the normal flow of instruction execution. It usually indicates a failure.

A **condition value** is a 32-bit value used to identify uniquely an exception condition. A condition value can be returned to a calling program as a function value or signaled using the VAX signaling mechanism.

Language support procedures are called implicitly to implement higher-level language constructs. They are not intended to be called explicitly from user programs.

Library procedures are called explicitly using the equivalent of a CALL statement or function reference. They are usually language independent.

2.2 Calling Sequence

At the option of the calling program, you invoke the called procedure using either the CALLG or CALLS instruction, as follows:

```
CALLG    arglst, proc
CALLS    argcnt, proc
```

CALLS pushes the argument count **argcnt** onto the stack as a longword and sets the argument pointer, AP, to the top of the stack. The complete sequence using CALLS is as follows:

```
push     argn
  .
  .
  .
push     arg1
CALLS    #n, proc
```

If the called procedure returns control to the calling program, control must return to the instruction immediately following the CALLG or CALLS instruction. Skip returns and GOTO returns are allowed only during stack unwind operations.

The called procedure returns control to the calling program by executing the return instruction, RET.

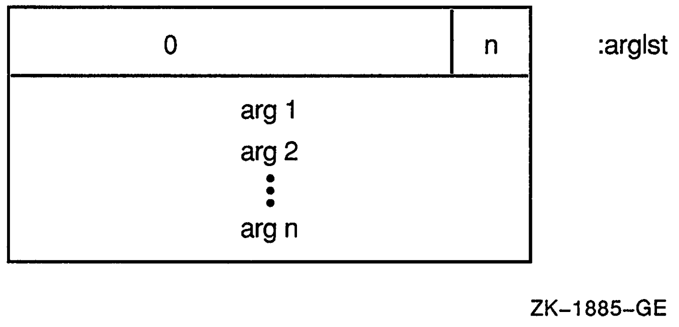
2.3 Argument List

The argument list is the primary means of passing information to and receiving results from a procedure.

2.3.1 Argument List Format

Figure 2–1 shows the argument list format.

Figure 2–1 Argument List Format



The first longword is always present and contains the argument count as an unsigned integer in the low byte. The 24 high-order bits are reserved by Digital and must be zero. To access the argument count, the called procedure must ignore the reserved bits and access the count as an unsigned byte (for example, MOVZBL, TSTB, or CMPB).

The remaining longwords can be one of the following:

- An uninterpreted 32-bit value (by immediate value mechanism). If the called procedure expects fewer than 32 bits, it accesses the low-order bits and ignores the high-order bits.
- An address (by reference mechanism). It is typically a pointer to a scalar data item, an array, a structure, a record, or a procedure.
- An address of a descriptor (by descriptor mechanism). See Section 2.9 for descriptor formats.

The standard permits programs to call by immediate value, by reference, by descriptor, or combinations of these mechanisms. Interpretation of each argument list entry depends on agreement between the calling and called procedures. Higher-level languages use the reference or descriptor mechanisms for passing input parameters. VMS system services and VAX BLISS, VAX C, or VAX MACRO programs use all three mechanisms.

A procedure with no arguments is called with a list consisting of a 0 argument count longword, as follows:

```
CALLS    #0, proc
```

A missing or null argument—for example, CALL SUB(A,,B)—is represented by an argument list entry consisting of a longword 0. Some procedures allow trailing null arguments to be omitted, others require all arguments. See each procedure's specification for details.

The argument list must be treated as read-only data by the called procedure and can be allocated in read-only memory at the option of the calling program.

2.3.2 Argument Lists and Higher-Level Languages

Functional notations for procedure calls in higher-level languages are mapped into VAX argument lists according to the following rules:

- Arguments are mapped from left to right to increasing argument list offsets. The leftmost (first) argument has an address of **arglst+4**; the next has an address of **arglst+8**, and so on. The only exception to this is when **arglst+4** specifies where a function value is to be returned, in which case the first argument has an address of **arglst+8**; the second argument has an address of **arglst+12**, and so on. See Section 2.4 for more information.
- Each argument position corresponds to a single VAX argument list entry.

2.3.2.1 Order of Argument Evaluation

Because most higher-level languages do not specify the order of evaluation of arguments (with respect to side effects), those language processors can evaluate arguments in any convenient order.

In constructing an argument list on the stack, a language processor can evaluate arguments from right to left and push their values on the stack. If call-by-reference semantics are used, argument expressions can be evaluated from left to right, with pointers to the expression values or descriptors being pushed from right to left.

The choice of argument evaluation order and code generation strategy is constrained only by the definition of the particular language. You should not write programs that depend on the order of evaluation of arguments.

2.3.2.2 Language Extensions for Argument Transmission

The VAX Procedure Calling Standard permits arguments to be passed by immediate value, by reference, or by descriptor. By default, all language processors, except VAX BLISS, VAX C, and VAX MACRO pass arguments by reference or by descriptor.

Language extensions are needed to reconcile the different argument-passing mechanisms. In addition to the default passing mechanism used, each language processor is required to give you explicit control, in the calling program, of the argument-passing mechanism for the data types supported by the language.

The value Yes means the language processor is required to provide the user explicit control of that passing mechanism.

Data Type	Section	Value	Reference	Descriptor
Atomic <= 32 bits	2.8.1	Yes	Yes	Yes
Atomic > 32 bits	2.8.1	No	Yes	Yes
String	2.8.2	No	Yes	Yes

Data Type	Section	Value	Reference	Descriptor
Miscellaneous	2.8.3	No[1]	No	No
Array	2.9	No	Yes	Yes

[1] For those languages supporting the **bound procedure value** data type, a language extension is required to pass it by immediate value in order to be able to interface with VMS system services and other software. See Section 2.8.3

For example, VAX FORTRAN provides the following intrinsic compile-time functions:

%VAL(arg)	By immediate value mechanism. Corresponding argument list entry is the 32-bit value of the argument, **arg**, as defined in the language.
%REF(arg)	By reference mechanism. Corresponding argument list entry contains the address of the value of the argument, **arg**, as defined in the language.
%DESCR(arg)	By descriptor mechanism. Corresponding argument list entry contains the address of a VAX descriptor of the argument, **arg**, as defined in Section 2.9 and in the language.

You can use these intrinsic functions in the syntax of a procedure call to control generation of the argument list. For example:

```
CALL SUB1(%VAL(123), %REF(X), %DESCR(A))
```

In other languages the same effect might be achieved by making appropriate attributes of the declaration of SUB1 in the calling program. Thus, you might write the following after making the external declaration for SUB1:

```
CALL SUB1 (123, X, A)
```

2.4 Function Value Return

A function value is returned in register R0 if its data type can be represented in 32 bits, or in registers R0 and R1 if its data type can be represented in 64 bits, provided the data type is not a string data type (see Section 2.8.2).

If the data type requires fewer than 32 bits, then R1 and the high-order bits of R0 are undefined. If the data type requires 32 or more bits but fewer than 64 bits, then the high-order bits of R1 are undefined. Two separate 32-bit entities cannot be returned in R0 and R1 because higher-level languages cannot process them.

In all other cases (the function value needs more than 64 bits, the data type is a string, the size of the value can vary from call to call, and so on) the actual argument list and the formal argument list are shifted one entry. The new, first entry is reserved for the function value. In this case, one of the following mechanisms is used to return the function value:

- If the maximum length of the function value is known (for example, octaword integer, H_floating, or fixed-length string), the calling program can allocate the required storage and pass the address of the storage or a descriptor for the storage as the first argument.

- If the maximum length of a string function value is not known to the calling program, the calling program can allocate a dynamic string descriptor. The called procedure then allocates storage for the function value and updates the contents of the dynamic string descriptor using VAX Run-Time Library procedures. See Section 2.9.3.

Some procedures, such as operating system calls and many library procedures, return a success or failure value as a longword function value in R0. Bit <0> of the value is set (Boolean true) for a success and clear (Boolean false) for a failure. The particular success or failure status is encoded in the remaining 31 bits, as described in Section 2.5.

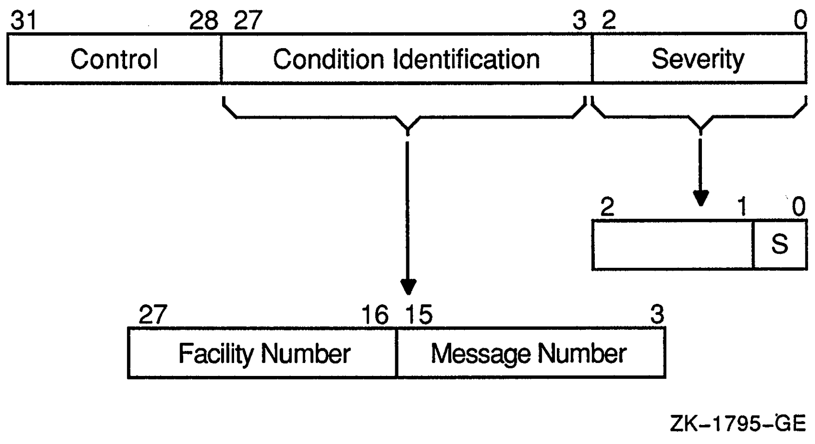
2.5 Condition Value

The VAX architecture uses condition values for the following reasons:

- To indicate the success or failure of a called procedure as a function value
- To describe an exception condition when an exception is signaled
- To identify system messages
- To report program success or failure to the command language level

A condition value is a longword that includes fields to describe the software component generating the value, the reason the value was generated, and the error severity status. Figure 2–2 shows the format of the condition value.

Figure 2–2 Format of the Condition Value



Fields in the Condition Value

condition identification
Identifies the condition uniquely on a systemwide basis.

facility
Identifies the software component generating the condition value. Bit <27> is set for customer facilities and clear for Digital facilities.

message number
Describes the status, which can be a hardware exception that occurred or a software-defined value. Message numbers with bit <15> set are specific to a single facility. Message numbers with bit <15> clear are systemwide status codes.

severity
Indicates success or failure. The severity code bit <0> is set for success (logical true) and clear for failure (logical false); bits <1> and <2> distinguish degrees of success or failure. Bits <2:0>, when taken as an unsigned integer, are interpreted as shown in the following table.

Symbol	Value	Description
STS$K_WARNING	0	Warning
STS$K_SUCCESS	1	Success
STS$K_ERROR	2	Error
STS$K_INFO	3	Information
STS$K_SEVERE	4	Severe_error
	5	Reserved by Digital
	6	Reserved by Digital
	7	Reserved by Digital

Section 2.5.1 describes the severity code more fully.

cntrl
Controls the printing of the message associated with the condition value. Bit <28> inhibits the message associated with the condition value from being printed by the SYS$EXIT system service. This bit is set by the system default handler after it has output an error message using the SYS$PUTMSG system service. It should also be set in the condition value returned by a procedure as a function value, if the procedure has also signaled the condition (so that the condition has been either printed or suppressed). Bits <31:29> must be zero; they are reserved by Digital for future use.

Software symbols are defined for these fields, as follows:

Symbol	Value	Meaning	Field
STS$V_COND_ID	3	Position of 27:3	Condition identification
STS$S_COND_ID	25	Size of 27:3	Condition identification
STS$M_COND_ID	Mask	Mask for 27:3	Condition identification
STS$V_INHIB_MSG	1@28	Position for 28	Inhibit message on image exit

Symbol	Value	Meaning	Field
STS$S_INHIB_MSG	1	Size for 28	Inhibit message on image exit
STS$M_INHIB_MSG	Mask	Mask for 28	Inhibit message on image exit
STS$V_FAC_NO	16	Position of 27:16	Facility number
STS$S_FAC_NO	12	Size of 27:16	Facility number
STS$M_FAC_NO	Mask	Mask for 27:16	Facility number
STS$V_CUST_DEF	27	Position for 27	Customer facility
STS$S_CUST_DEF	1	Size for 27	Customer facility
STS$M_CUST_DEF	1@27	Mask for 27	Customer facility
STS$V_MSG_NO	3	Position of 15:3	Message number
STS$S_MSG_NO	13	Size of 15:3	Message number
STS$M_MSG_NO	Mask	Mask for 15:3	Message number
STS$V_FAC_SP	15	Position of 15	Facility specific
STS$S_FAC_SP	1	Size for 15	Facility specific
STS$M_FAC_SP	1@15	Mask for 15	Facility specific
STS$V_CODE	3	Position of 14:3	Message code
STS$S_CODE	12	Size of 14:3	Message code
STS$M_CODE	Mask	Mask for 14:3	Message code
STS$V_SEVERITY	0	Position of 2:0	Severity
STS$S_SEVERITY	3	Size of 2:0	Severity
STS$M_SEVERITY	7	Mask for 2:0	Severity
STS$V_SUCCESS	0	Position of 0	Success
STS$S_SUCCESS	1	Size of 0	Success
STS$M_SUCCESS	1	Mask for 0	Success

2.5.1 Interpretation of Severity Codes

A severity code of 0 indicates a warning. This code is used whenever a procedure produces output but the output produced might not be what the user expected—for example, a compiler modification of a source program.

A severity code of 1 indicates that the procedure generating the condition value completed successfully, as expected.

A severity code of 2 indicates that an error has occurred but the procedure did produce output. Execution can continue, but the results produced by the component generating the condition value are not all correct.

A severity code of 3 indicates that the procedure generating the condition value completed successfully but has some parenthetical information to be included in a message if the condition is signaled.

A severity code of 4 indicates that a severe error occurred and the component generating the condition value was unable to produce output.

When designing a procedure, you should base the choice of severity code for its condition values on the following default interpretations:

- The calling program typically performs a low-bit test, so it treats warnings, errors, and severe errors as failures, and success and information as successes.
- If the condition value is signaled (see Section 2.11.3), the default handler treats severe errors as reason to terminate and all the others as the basis for attempting to continue.
- When the program image exits, the command interpreter by default treats errors and severe errors as the basis for stopping the job, and warnings, information, and successes as the basis for continuing.

The following table summarizes the default interpretation of condition values.

Severity	Routine	Signal	Default at Program Exit
Success	Normal	Continue	Continue
Information	Normal	Continue	Continue
Warning	Failure	Continue	Continue
Error	Failure	Continue	Stop job
Severe error	Failure	Exit	Stop job

The default for signaled messages is to output a message to SYS$OUTPUT. In addition, for severities other than success (STS$K_SUCCESS), a copy of the message is made on SYS$ERROR. At program exit, success and information completion values do not generate messages; however, warning, error, and severe error condition values do generate messages to both SYS$OUTPUT and SYS$ERROR, unless bit <28> (STS$V_INHIB_MSG) is set.

Unless there is a good basis for another choice, a procedure should use either success or severe error as its severity for each condition value.

2.5.2 Use of Condition Values

VAX software components return condition values when they complete execution. When a severity code of warning, error, or severe error is generated, the status code describes the nature of the problem. This value can be tested to change the flow of control of a procedure or be used to generate a message, or both.

User procedures can also generate condition values to be examined by other procedures and by the command interpreter. User-generated condition values should have bits <27> and <15> set so that they do not conflict with values generated by Digital.

2.6 Register Usage

Scalar and vector register usage rules are considered separately.

2.6.1 Scalar Register Usage

The following registers have defined uses:

Register	Use
PC	Program counter.
SP	Stack pointer.
FP	Current stack frame pointer. It must always point at the current frame. No modification is permitted within a procedure body.
AP	Argument pointer. When a call occurs, AP must point to a valid argument list. A procedure without parameters points to an argument list consisting of a single longword containing the value *0*.
R1	Environment value. When a procedure that needs an environment value is called, the calling program must set R1 to the environment value. See bound procedure value in Section 2.8.3.
R0,R1	Function value return registers. These registers are not to be preserved by any called procedure. They are available as temporary registers to any called procedure.

Registers R2 through R11 are to be preserved across procedure calls. The called procedure can use these registers, provided it saves and restores them using the procedure entry mask mechanism. The entry mask mechanism must be used so that any stack unwinding done by the condition-handling mechanism restores all registers correctly. In addition, PC, SP, FP, and AP are always preserved by the CALL instruction and restored by the RET instruction. However, a called procedure can use AP as a temporary register.

If JSB routines are used, they must not save or modify any preserved registers (R2 through R11) not already saved by the entry mask mechanism of the calling program.

2.6.2 Vector Register Usage

The VAX Calling Standard specifies no conventions for preserved vector registers, vector argument registers, or vector function value return registers. All such conventions are by agreement between the calling and called procedures. In the absence of such an agreement, all vector registers, including V0 through V15, VLR, VCR, and VMR are scratch registers. Among cooperating procedures, a procedure that does preserve or otherwise manipulate the vector registers by agreement with its callers must provide an exception handler to restore them during an unwind.

2.6.3 Vector and Scalar Processor Synchronization

There are two kinds of synchronization between a scalar and vector processor pair: memory synchronization and exception synchronization.

2.6.3.1 Memory Synchronization

Every procedure is responsible for synchronization of memory operations with the calling procedure and with procedures it calls. If a procedure executes vector loads or stores, one of the following must occur:

- An MSYNC instruction (a form of the MFVP instruction) must be executed before the first vector load/store to synchronize with memory operations issued by the caller. While an MSYNC instruction might typically occur in the entry code sequence of a procedure, exact placement might also depend on a variety of optimization considerations.
- An MSYNC instruction must be executed after the last vector load or store to synchronize with memory operations issued after return. While an MSYNC instruction might typically occur in the return code sequence of a procedure, exact placement might also depend on a variety of optimization considerations.
- An MSYNC must be executed between each vector load/store and each standard call to other procedures to synchronize with memory operations issued by those procedures.

That is, any procedure that executes vector loads or stores is responsible for synchronizing with potentially conflicting memory operations in any other procedure. However, execution of an MSYNC to ensure scalar/vector memory synchronization can be omitted when it can be determined for the current procedure that all possibly incomplete vector load/stores operate only on memory that is not accessed by other procedures.

2.6.3.2 Exception Synchronization

Every procedure must ensure that no exception can be raised after the current frame is changed (as a result of either a CALL or RET). If a procedure executes any vector instruction that might possibly raise an exception, then a SYNC instruction (a form of the MFVP instruction) must be executed prior to any subsequent CALL or RET.

However, if the only possible exceptions that can occur are ensured to be reported by an MSYNC instruction that is otherwise needed for memory synchronization, then the SYNC is redundant and can be omitted as an optimization.

Moreover, if the only possible exceptions that can occur are ensured to be reported by one or more MFVP instructions that read the vector control registers, then the SYNC is redundant and can be omitted as an optimization.

2.6.3.3 Synchronization Summary

Memory synchronization with the caller of a procedure that uses the vector processor is required because there might be scalar machine writes (to main memory) still pending at the time of entry to the called procedure. The various forms of write-cache strategies allowed by the VAX architecture combined with the possibly independent scalar and vector memory access paths implies that a scalar store followed by a CALL followed by a vector load is not safe without an intervening MSYNC.

Within a procedure that uses the vector processor, proper memory and exception synchronization might require use of an MSYNC instruction or a SYNC instruction, or both, prior to calling another procedure or upon being called by another procedure. Further, for calls to other procedures, the requirements can vary from call to call, depending on details of actual vector usage.

An MSYNC instruction (without a SYNC) at procedure entry, at procedure exit, and prior to a call provides proper synchronization in most cases. A SYNC instruction without an MSYNC prior to a CALL (or RET) is sometimes appropriate. The remaining two cases, where either both or neither MSYNC and SYNC are needed, are probably rare.

Refer to the VAX Vector Architecture section in the *VAX MACRO and Instruction Set Reference Manual* for the specific rules on what exceptions are ensured to be reported by MSYNC and other MFVP instructions.

2.7 Stack Usage

Figure 2–3 shows the contents of the stack frame that is created for the called procedure by the CALLG or CALLS instruction.

Figure 2–3 Stack Frame Generated by CALLG and CALLS Instructions

```
condition handler (0)                 :(SP):(FP))
mask/PSW
AP
FP
PC
R2    (optional)
.
.
.
R11   (optional)
```

FP always points at the condition handler longword of the stack frame. Other uses of FP within a procedure are prohibited. See Section 2.10 for more information.

The contents of the stack located at addresses higher than the mask /PSW longword belong to the calling program; they should not be read or written by the called procedure, except as specified in the argument list. The contents of the stack located at addresses lower than SP belong to interrupt and exception routines; they are modified continually and unpredictably.

The called procedure allocates local storage by subtracting the required number of bytes from the SP provided on entry. This local storage is freed automatically by the RET instruction.

Bit <28> of the mask/PSW longword is reserved by Digital for future extensions to the stack frame.

2.8 Argument Data Types

Each data type implemented for a higher-level language uses one of the following VAX data types for procedure parameters and elements of file records:

- Atomic
- String
- Miscellaneous

When existing data types are not sufficient to satisfy the semantics of a language, new data types are added to this standard, including certain language-specific ones. These data types can generally be passed by immediate value (if 32 bits or less), by reference, or by descriptor.

You use the encoding given in Sections 2.8.1 and 2.8.2 whenever you need to identify data types, such as in a descriptor. However, in addition to their use in descriptors, these data type codes are also useful in areas outside the scope of the Procedure Calling Standard for identifying VAX data types. Therefore, each data type code indicates a unique data format independent of its use in descriptors.

Some data types are composed of a record-like structure consisting of two or more elementary data types. For example, the F_floating complex (FC) data type is made up of two F_floating data types, and the varying character string (VT) data type is made up of a word (unsigned, WU) data type followed by a character string (T) data type.

Unless stated otherwise, all data types represent signed quantities. The unsigned quantities throughout this standard do not allocate space for the sign; all bit or character positions are used for significant data.

2.8.1 Atomic Data Types

Table 2–1 shows how atomic data types are defined and encoded.

Table 2–1 Atomic Data Types

Symbol	Code	Name/Description
DSC$K_DTYPE_Z	0	Unspecified The calling program has specified no data type. The default argument for the called procedure should be the correct type.
DSC$K_DTYPE_BU	2	Byte (unsigned) 8-bit unsigned quantity.
DSC$K_DTYPE_WU	3	Word (unsigned) 16-bit unsigned quantity.
DSC$K_DTYPE_LU	4	Longword (unsigned) 32-bit unsigned quantity.
DSC$K_DTYPE_QU	5	Quadword (unsigned) 64-bit unsigned quantity.
DSC$K_DTYPE_OU	25	Octaword (unsigned) 128-bit unsigned quantity.
DSC$K_DTYPE_B	6	Byte integer (signed) 8-bit signed 2's-complement integer.
DSC$K_DTYPE_W	7	Word integer (signed) 16-bit signed 2's-complement integer.
DSC$K_DTYPE_L	8	Longword integer (signed) 32-bit signed 2's-complement integer.
DSC$K_DTYPE_Q	9	Quadword integer (signed) 64-bit signed 2's-complement integer.
DSC$K_DTYPE_O	26	Octaword integer (signed) 128-bit signed 2's-complement integer.
DSC$K_DTYPE_F	10	F_floating 32-bit F_floating quantity representing a single-precision number.
DSC$K_DTYPE_D	11	D_floating 64-bit D_floating quantity representing a double-precision number.
DSC$K_DTYPE_G	27	G_floating 64-bit G_floating quantity representing a double-precision number.
DSC$K_DTYPE_H	28	H_floating 128-bit H_floating quantity representing a quadruple-precision number.

(continued on next page)

Table 2–1 (Cont.) Atomic Data Types

Symbol	Code	Name/Description
DSC$K_DTYPE_FC	12	F_floating complex Ordered pair of F_floating quantities, representing a single-precision complex number. The lower addressed quantity is the real part, the higher addressed quantity is the imaginary part.
DSC$K_DTYPE_DC	13	D_floating complex Ordered pair of D_floating quantities, representing a double-precision complex number. The lower addressed quantity is the real part, the higher addressed quantity is the imaginary part.
DSC$K_DTYPE_GC	29	G_floating complex Ordered pair of G_floating quantities, representing a double-precision complex number. The lower addressed quantity is the real part, the higher addressed quantity is the imaginary part.
DSC$K_DTYPE_HC	30	H_floating complex Ordered pair of H_floating quantities, representing a quadruple-precision complex number. The lower addressed quantity is the real part, the higher addressed quantity is the imaginary part.
DSC$K_DTYPE_CIT	31	COBOL Intermediate Temporary A floating-point datum with an 18-digit normalized decimal fraction and a 2-decimal-digit exponent. The fraction is a packed decimal string. The exponent is a 16-bit 2's-complement integer. See Section 2.8.6 for details.

2.8.2 String Data Types

String data types are ordinarily described by a string descriptor. Table 2–2 shows how the string data types are defined and encoded.

Table 2–2 String Data Types

Symbol	Code	Name/Description
DSC$K_DTYPE_T	14	Character string A single 8-bit character (atomic data type) or a sequence of 0 to 2^{16}-1 8-bit characters (string data type).

(continued on next page)

Table 2–2 (Cont.) String Data Types

Symbol	Code	Name/Description
DSC$K_DTYPE_VT	37	Varying character string A 16-bit unsigned count of the current number of 8-bit characters following, followed by a sequence of 0 to 2^{16}-1 8-bit characters (see Section 2.8.7 for details). When this data type is used with descriptors, it can only be used with the varying string and varying string array descriptors because the length field is interpreted differently from the other 8-bit string data types. (See Sections 2.8.7, 2.9.12, and 2.9.13 for further discussion.)
DSC$K_DTYPE_NU	15	Numeric string, unsigned
DSC$K_DTYPE_NL	16	Numeric string, left separate sign
DSC$K_DTYPE_NLO	17	Numeric string, left overpunched sign
DSC$K_DTYPE_NR	18	Numeric string, right separate sign
DSC$K_DTYPE_NRO	19	Numeric string, right overpunched sign
DSC$K_DTYPE_NZ	20	Numeric string, zoned sign
DSC$K_DTYPE_P	21	Packed-decimal string
DSC$K_DTYPE_V	1	Aligned bit string A string of 0 to 2^{16}-1 contiguous bits. The first bit is bit <0> of the first byte, and the last bit is any bit in the last byte. Remaining bits in the last byte must be zero on read and are cleared on write. Unlike the unaligned bit string (VU) data type, when the aligned bit string (V) data type is used in array descriptors, the ARSIZE field is in units of bytes, not bits, because allocation is a multiple of 8 bits.
DSC$K_DTYPE_VU	34	Unaligned bit string The data is 0 to 2^{16}-1 contiguous bits located arbitrarily with respect to byte boundaries. See also aligned bit string (V) data type. Because additional information is required to specify the bit position of the first bit, this data type can be used only with the unaligned bit string and unaligned bit array descriptors (see Sections 2.9.14 and 2.9.15).

2.8.3 Miscellaneous Data Types

Table 2–3 shows how miscellaneous data types are defined and encoded.

Table 2–3 Miscellaneous Data Types

Symbol	Code	Name/Description
DSC$K_DTYPE_ZI	22	Sequence of instructions
DSC$K_DTYPE_ZEM	23	Procedure entry mask
DSC$K_DTYPE_DSC	24	Descriptor This data type allows a descriptor to be a data type; thus, levels of descriptors are allowed.
DSC$K_DTYPE_BPV	32	Bound procedure value A 2-longword entity in which the first longword contains the address of a procedure entry mask and the second longword is the environment value. The environment value is determined in a language-specific manner when the original bound procedure value is generated. When the bound procedure is called, the calling program loads the second longword into R1. When the environment value is not needed, this data type can be passed using the immediate value mechanism. In this case, the argument list entry contains the address of the procedure entry mask and the second longword is omitted.
DSC$K_DTYPE_BLV	33	Bound label value A 2-longword entity in which the first longword contains the address of an instruction and the second longword is the language-specific environment value. The environment value is determined in a language-specific manner when the original bound label value is generated.
DSC$K_DTYPE_ADT	35	Absolute date and time A 64-bit unsigned, scaled, binary integer representing a date and time in 100-nanosecond units offset from the VMS operating system base date and time, which is 00:00 o'clock, November 17, 1858 (the Smithsonian base date and time for astronomical calendars). The value zero indicates that the date and time have not been specified, so a default value or distinctive print format can be used. Note that the ADT data type is the same as the VMS date format for positive values only.

2.8.4 Facility-Specific Data Type Codes

Data type codes 160 through 191 are reserved by Digital for facility-specific purposes. These codes must not be passed between facilities because different facilities can use the same code for different purposes. These codes might be used by compiler-generated code to pass parameters to the language-specific run-time support procedures associated with that language or the VMS Debugger.

2.8.5 Reserved Data Type Codes

The type codes 38 through 191 are reserved by Digital. Codes 192 through 255 are reserved for Digital's Computer Special Systems Group and for customers for their own use.

2.8.6 COBOL Intermediate Temporary Data Type

A COBOL intermediate temporary datum is 12 contiguous bytes starting on an arbitrary byte boundary. It is specified by its address, A, as follows:

15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0	
Exponent				:A
f<16>	f<15>	0	f<17>	:A+2
f<12>	f<11>	f<14>	f<13>	:A+4
f<8>	f<7>	f<10>	f<9>	:A+6
f<4>	f<3>	f<6>	f<5>	:A+8
f<0>	Sign	f<2>	f<1>	:A+10

ZK–1887–GE

A COBOL intermediate temporary datum represents a floating-point datum with a normalized, 18-digit, packed-decimal fraction and a 16-bit 2's-complement integer exponent. Bytes 0 and 1 are the exponent. Bytes 2 through 11 contain the normalized packed-decimal fraction. The sign of the datum is the sign of the fraction. If the fraction is zero, the value of the datum is zero.

If the exponent is from *–99* to *+99*, operations can be performed on this datum. If the exponent is outside this range, a reserved operand condition is signaled (see Section 2.12). If a calculated datum has an exponent greater than *+99*, the exact result with the low-order 15 bits of the true exponent is stored in the result datum and an overflow condition is signaled.

If a calculated datum has an exponent less than *–99*, the exact result with the low-order 15 bits of the true exponent is stored in the result datum and an underflow condition is signaled. The condition handler can take the appropriate action. Condition mnemonics have a COB$_ prefix and are documented with the COBOL part of the VMS Run-Time Library. An exponent value of *–32768* is taken as reserved and should be used to encode reserved operands such as uninitialized datum and indeterminate value. By convention, if the fraction of a result is zero, the exponent is set to zero. Fractions are generated with preferred sign codes and avoid minus zero.

2.8.7 Varying Character String Data Type (DSC$K_DTYPE_VT)

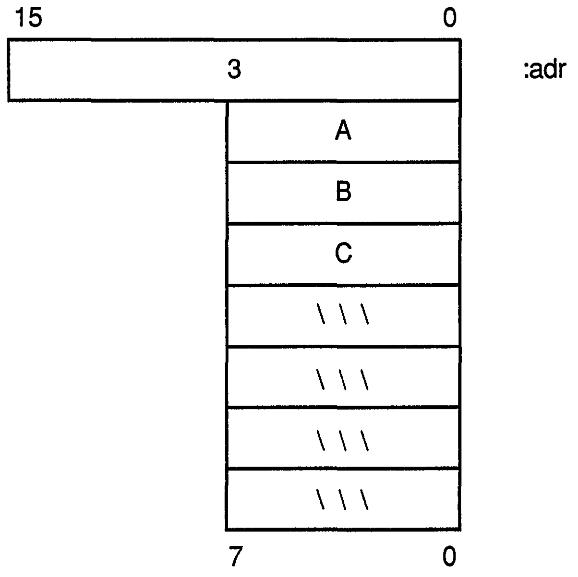
The varying character string data type consists of the following two fixed-length areas allocated contiguously with no padding in between:

CURLEN	An unsigned word specifying the current length in bytes of the immediately following string (byte aligned).
BODY	A fixed-length area containing the string that can vary from zero to a maximum length defined for each instance of string. The range of this maximum length is 0 to 2^{16}-1.

When passed by reference or by descriptor, the address of the varying character string (VT) data type is always the address of the CURLEN field, not the BODY field.

When a called procedure modifies a varying character string data type passed by reference or by descriptor, it writes the new length, *n*, into CURLEN and can modify all bytes of BODY—even those beyond the new length.

For example, consider a varying string with a maximum length of seven characters. For the representation of the string, ABC, CURLEN would be three and the last four bytes would be undefined, as follows:



ZK–1889–GE

2.9 Argument Descriptor Formats

A uniform descriptor mechanism is defined for use by all procedures that conform to the VAX Procedure Calling Standard. Descriptors are self describing, and the mechanism is extensible. When existing descriptors are not sufficient to satisfy the semantics of a language, new descriptors are added to this standard.

Unless stated otherwise, the calling program fills in all fields in descriptors. This is true whether the descriptor is generated by default or by a language extension. The fields are filled in even if a called procedure written in the same language would ignore the contents of some of the fields.

A descriptor conforms to the VAX Procedure Calling Standard if all fields are filled in by the calling program according to the standard, even if the field is not needed by the called program.

Note: Unless stated otherwise, all fields in descriptors represent unsigned quantities, are read-only from the point of view of the called procedure, and can be allocated in read-only memory at the option of the calling program.

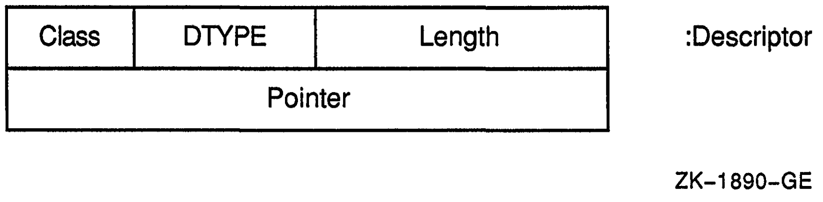
If a language processor implements a language-specific data type that is not added to this standard (see Section 2.8), it is not required to use a standard descriptor to pass an array of such a data type. However, if a language processor does pass an array of such a data type using a standard descriptor, the language processor fills in the DSC$B_DTYPE field with zero, indicating that the data type field is unspecified, rather than using a more general data type code.

For example, an array of PL/I POINTER data types has the DTYPE field filled in with the value *0* (unspecified data type) rather than with the value *4* (longword (unsigned) data type). The remaining fields are filled in as specified by this standard; for example, DSC$W_LENGTH is filled in with the size in bytes. Because the language-specific data type might be added to the standard in the future, generic application procedures that examine the DTYPE field should be prepared for zero and for additional data types.

2.9.1 Descriptor Prototype

Figure 2–4 shows the descriptor prototype format, which consists of at least two longwords.

Figure 2–4 Descriptor Prototype Format



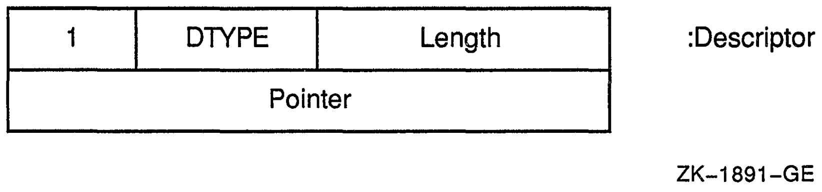
Symbol	Description
DSC$W_LENGTH <0,15:0>	A 1-word field specific to the descriptor class, typically a 16-bit (unsigned) length.
DSC$B_DTYPE <0,23:16>	A 1-byte data type code. Data type codes are listed in Sections 2.8.1 and 2.8.2.
DSC$B_CLASS <0,31:24>	A 1-byte descriptor class code that identifies the format and interpretation of the other fields of the descriptor as specified in the following sections. This interpretation is intended to be independent of the DTYPE field, except for the data types that are made up of units that are less than a byte (packed-decimal string (P), aligned bit string (V), and unaligned bit string (VU)). The CLASS code can be used at run time by a called procedure to determine which descriptor is being passed.
DSC$A_POINTER <1,31:0>	A longword containing the address of the first byte of the data element described.

Note that the descriptor can be placed in a pair of registers with a MOVQ instruction and then the length and address can be used directly. This gives a word length, so the class and type are placed as bytes in the rest of that longword. When the class field is zero, no more than the preceding information can be assumed.

2.9.2 Fixed-Length Descriptor (DSC$K_CLASS_S)

A single descriptor form is used for scalar data and fixed-length strings. Any VAX data type can be used with this descriptor, except data type 34 (unaligned bit string). Figure 2–5 shows the format of a fixed-length descriptor.

Figure 2–5 Fixed-Length Descriptor Format



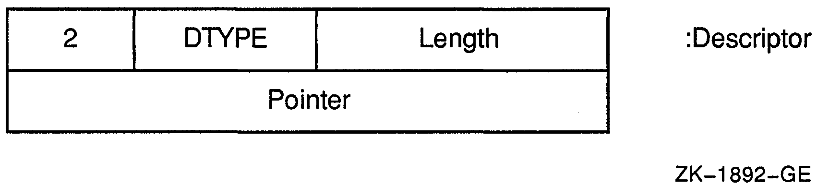
Symbol	Description
DSC$W_LENGTH	Length of data item in bytes, unless the DSC$B_DTYPE field contains the value *1* (aligned bit string) or *21* (packed-decimal string). Length of data item is in bits for bit. Length of data item is the number of 4-bit digits (not including the sign) for packed-decimal string.
DSC$B_DTYPE	A 1-byte data type code. Data type codes are listed in Sections 2.8.1 and 2.8.2.
DSC$B_CLASS	1 = DSC$K_CLASS_S.
DSC$A_POINTER	Address of first byte of data storage.

If the data type is 14 (character string) and the string must be extended in a string comparison or is being copied to a fixed-length string containing a greater length, the space character (hexadecimal 20 if ASCII) is used as the fill character.

2.9.3 Dynamic String Descriptor (DSC$K_CLASS_D)

A single descriptor form is used for dynamically allocated strings. When a string is written, either or both the length field and the pointer field can be changed. The VMS Run-Time Library provides procedures for changing fields. As an input parameter, this format is interchangeable with class 1 (DSC$K_CLASS_S). Figure 2–6 shows the format of a dynamic string descriptor.

Figure 2–6 Dynamic String Descriptor Format



Symbol	Description
DSC$W_LENGTH	Length of data item in bytes, unless the DSC$B_DTYPE field contains the value *1* (aligned bit string) or *21* (packed-decimal string). Length of data item is in bits for bit. Length of data item is the number of 4-bit digits (not including the sign) for packed-decimal string.
DSC$B_DTYPE	A 1-byte data type code. Data type codes are listed in Sections 2.8.1 and 2.8.2.
DSC$B_CLASS	2 = DSC$K_CLASS_D.
DSC$A_POINTER	Address of first byte of data storage.

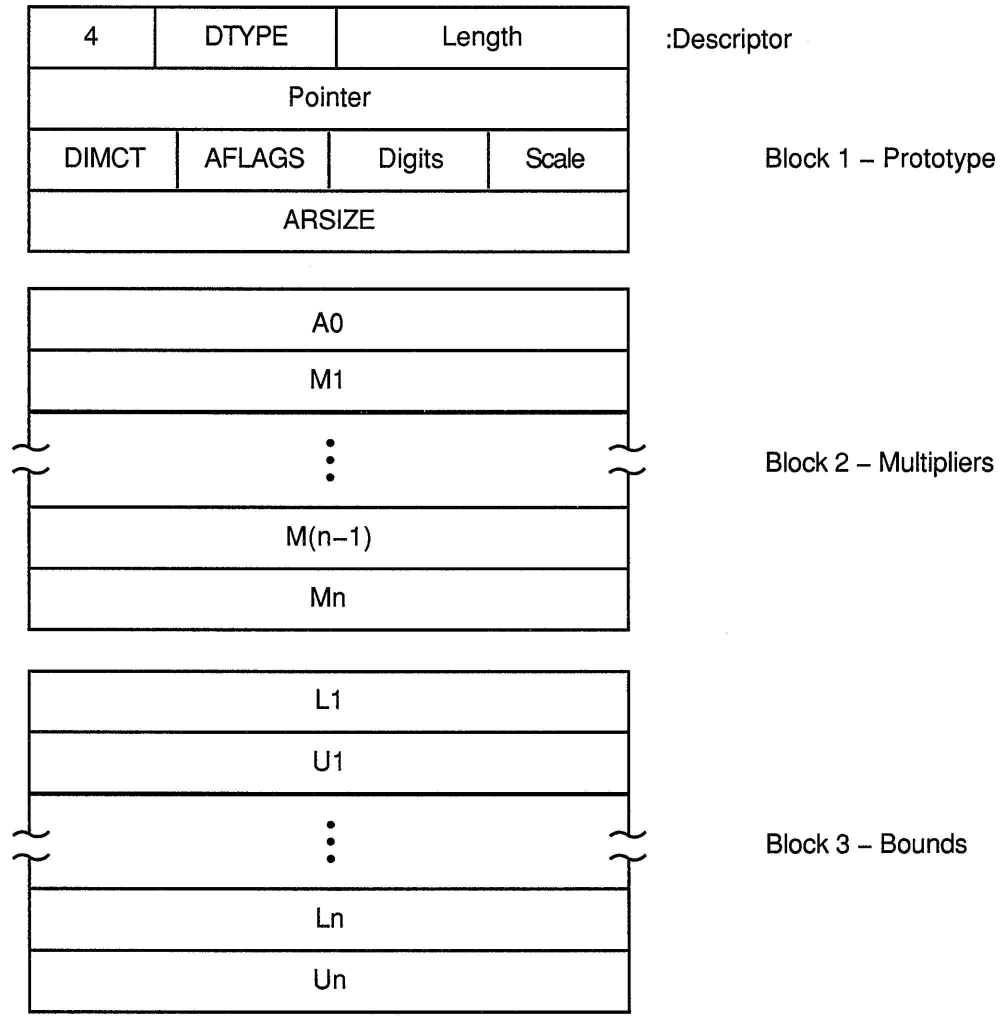
2.9.4 Variable Buffer Descriptor (DSC$K_CLASS_V)

This descriptor is reserved for use by Digital.

2.9.5 Array Descriptor (DSC$K_CLASS_A)

The array descriptor is used to describe contiguous arrays of atomic data types or contiguous arrays of fixed-length strings. An array descriptor consists of three contiguous blocks. The first block contains the descriptor prototype information and is part of every array descriptor. The second and third blocks are optional. If the third block is present, so is the second. Figure 2–7 shows the format of an array descriptor.

Figure 2–7 Array Descriptor Format



ZK–1888–GE

Symbol	Description
DSC$W_LENGTH	Length of an array element in bytes, unless the SC$B_DTYPE field contains the value *1* (aligned bit string) or *21* (packed-decimal string). Length of an array element is in bits for bit. Length of an array element is the number of 4-bit digits (not including the sign) for packed-decimal string.
DSC$B_DTYPE	A 1-byte data type code. Data type codes are listed in Sections 2.8.1 and 2.8.2.
DSC$B_CLASS	4 = DSC$K_CLASS_A.
DSC$A_POINTER	Address of first actual byte of data storage.

Symbol	Description	
DSC$B_SCALE <2,7:0>	Signed power-of-two or -ten multiplier, as specified by DSC$V_FL_BINSCALE, to convert the internal form to external form. (See Section 2.9.10.)	
DSC$B_DIGITS <2,15:8>	If nonzero, the unsigned number of decimal digits in the internal representation. If zero, the number of digits can be computed based on DSC$W_LENGTH. This field should be zero unless the DSC$B_TYPE field specifies a string data type that could contain numeric values.	
DSC$B_AFLAGS <2,23:16>	Array flag bits:	
	Reserved <2,18:16>	Must be zero.
	DSC$V_FL_BINSCALE <2,19>	If set, the scale factor specified by DSC$B_SCALE is a signed power-of-two multiplier to convert the internal form to external form. If not set, DSC$B_SCALE specifies a signed power-of-ten multiplier. (See Section 2.9.10.)
	DSC$V_FL_REDIM <2,20>	If set, the array can be redimensioned; that is, DSCA_A0, DSCL_Mi,DSC$L_Li, and DSC$L_Ui can be changed. The redimensioned array cannot exceed the size allocated to the array DSC$L_ARSIZE.
	DSC$V_FL_COLUMN <2,21>	If set, the elements of the array are stored by columns (FORTRAN). That is, the leftmost subscript (first dimension) is varied most rapidly, and the rightmost subscript (nth dimension) is varied least rapidly. If not set, the elements are stored by rows (most other languages). That is, the rightmost subscript is varied most rapidly and the leftmost subscript is varied least rapidly.
	DSC$V_FL_COEFF <2,22>	If set, the multiplicative coefficients in Block 2 are present. Must be set if DSC$V_FL_BOUNDS is set.
	DSC$V_FL_BOUNDS <2,23>	If set, the bounds information in Block 3 is present and requires that DSC$V_FL_COEFF be set.
DSC$B_DIMCT <2,31:24>	Number of dimensions, n.	

Symbol	Description
DSC$L_ARSIZE <3,31:0>	Total size of array (in bytes unless the DSC$B_TYPE field contains the value *21*; see the description for DSC$W_LENGTH). A redimensioned array can use less than the total size allocated. For data type 1 (aligned bit string), DSC$W_LENGTH is in bits while DSC$L_ARSIZE is in bytes because the unit of length is bits while the unit of allocation is aligned bytes.
DSC$A_A0 <4,31:0>	Address of element A(0,0, . . . ,0). This need not be within the actual array. It is the same as DSC$A_POINTER for zero-origin arrays.
DSC$L_Mi <4+i,31:0>	Addressing coefficients (Mi = Ui-Li+1).
DSC$L_Li <3+n+2*i,31:0>	Lower bound (signed) of *i*th dimension.
DSC$L_Ui <4+n+2*i,31:0>	Upper bound (signed) of *i*th dimension.

The following formulas specify the effective address, E, of an array element.

Warning: Modification of the following formulas is required if DSC$B_DTYPE contains a 1 or 21 because DSC$W_LENGTH is given in bits or 4-bit digits rather than bytes.

The effective address, E, for element A(I):

$$\begin{aligned} E &= A_0 + I*LENGTH \\ &= POINTER + [I - L_1]*LENGTH \end{aligned}$$

The effective address, E, for element A(I_1,I_2) with DSC$V_FL_COLUMN clear:

$$\begin{aligned} E &= A_0 + [I_1*M_2 + I_2]*LENGTH \\ &= POINTER + [[I_1-L_1]*M_2 + I_2 - L_2]*LENGTH \end{aligned}$$

The effective address, E, for element A(I_1,I_2) with DSC$V_FL_COLUMN set:

$$\begin{aligned} E &= A_0 + [I_2*M_1 + I_1]*LENGTH \\ &= POINTER + [[I_2-L_2]*M_1 + I_1 - L_1]*LENGTH \end{aligned}$$

The effective address, E, for element A(I_1, . . . ,I_n) with DSC$V_FL_COLUMN clear:

$$\begin{aligned} E &= A_0 + [[[[...[I_1]*M_2 + ...]*M_{n-2} + I_{n-2}]*M_{n-1} \\ &\quad + I_{n-1}]*M_n + I_n]*LENGTH \\ &= POINTER + [[[[...[I_1 - L_1]*M_2 + ...]*M_{n-2} + I_{n-2} \\ &\quad - L_{n-2}]*M_{n-1} + I_{n-1} - \\ &L_{n-1}]*M_n + I_n - L_n]*LENGTH \end{aligned}$$

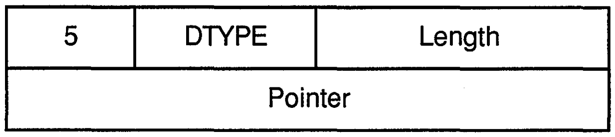
The effective address, E, for element A(I_1, . . . ,I_n) with DSC$V_FL_COLUMN set:

$$
\begin{aligned}
E = {} & A_0 + [[[[\ldots[I_n]*M_{n-1} + \ldots]*M_3 \\
& + I_3]*M_2 + I_2]*M_1 + I_1]*LENGTH \\
& = POINTER + [[[[\ldots[I_n - L_n]*M_{n-1} + \ldots]*M_3 + I_3 \\
& - L_3]*M_2 + I_2 - L_2]*M_1 \\
& + I_1 - L_1]*LENGTH
\end{aligned}
$$

2.9.6 Procedure Descriptor (DSC$K_CLASS_P)

The descriptor for a procedure specifies its entry address and function value data type, if any. Figure 2–8 shows the format of a procedure descriptor.

Figure 2–8 Procedure Descriptor Format



ZK–1893–GE

Symbol	Description
DSC$W_LENGTH	Length associated with the function value, or zero if no function value is returned.
DSC$B_DTYPE	Function value data type code. Data type codes are listed in Sections 2.8.1 and 2.8.2.
DSC$B_CLASS	5 = DSK$K_CLASS_P.
DSC$A_POINTER	Address of entry mask to routine.

Procedures return a function value in R0, R1/R0, or using the first argument list entry depending on the size of the data type (see Section 2.4).

2.9.7 Procedure Incarnation Descriptor (DSC$K_CLASS_PI)

The procedure incarnation descriptor is obsolete.

2.9.8 Label Descriptor (DSC$K_CLASS_J)

The label descriptor is reserved for use by the VMS Debugger.

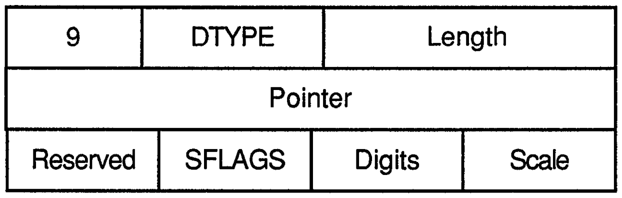
2.9.9 Label Incarnation Descriptor (DSC$K_CLASS_JI)

The label incarnation descriptor is obsolete.

2.9.10 Decimal String Descriptor (DSC$K_CLASS_SD)

Figure 2–9 shows the format of a decimal string descriptor. Decimal size and scaling information for both scalar data and simple strings is given in this descriptor form.

Figure 2–9 Decimal String Descriptor Format



ZK–1894–GE

Symbol	Description	
DSC$W_LENGTH	Length of data item in bytes, unless the DSC$B_DTYPE field contains the value *1* (aligned bit string) or *21* (packed-decimal string). Length of data item is in bits for bit. Length of data item is the number of 4-bit digits (not including the sign) for packed-decimal string.	
DSC$B_DTYPE	A 1-byte data type code. Data type codes are listed in Sections 2.8.1 and 2.8.2.	
DSC$B_CLASS	9 = DSC$K_CLASS_SD.	
DSC$A_POINTER	Address of first byte of data storage.	
DSC$B_SCALE <2,7:0>	Signed power-of-two or -ten multiplier, as specified by DSC$V_ FL_BINSCALE, to convert the internal form to external form. (See examples in the following table.)	
DSC$B_DIGITS <2,15:8>	If nonzero, the unsigned number of decimal digits in the internal representation. If zero, the number of digits can be computed based on DSC$W_LENGTH. This field should be zero unless the DSC$B_TYPE field specifies a string data type that could contain numeric values.	
DSC$B_SFLAGS <2,23:16>	Scalar flag bits:	
	Reserved <2,18:16>	Must be zero.
	DSC$V_FL_BINSCALE <2,19>	If set, the scale factor specified by DSC$B_SCALE is a signed power-of-two multiplier to convert the internal form to external form. If not set, DSC$B_SCALE specifies a signed power-of-ten multiplier. (See examples in the following table.)
	Reserved <2,23:20>	Must be zero.

Examples of DSC$B_SCALE and DSC$V_FL_BINSCALE interpretation are presented in the following table:

Internal Value	DSC$B_ SCALE	DSC$V_FL_ BINSCALE	External Value
123	+1	0	1230
123	+1	1	246
200	-2	0	2
200	-2	1	50

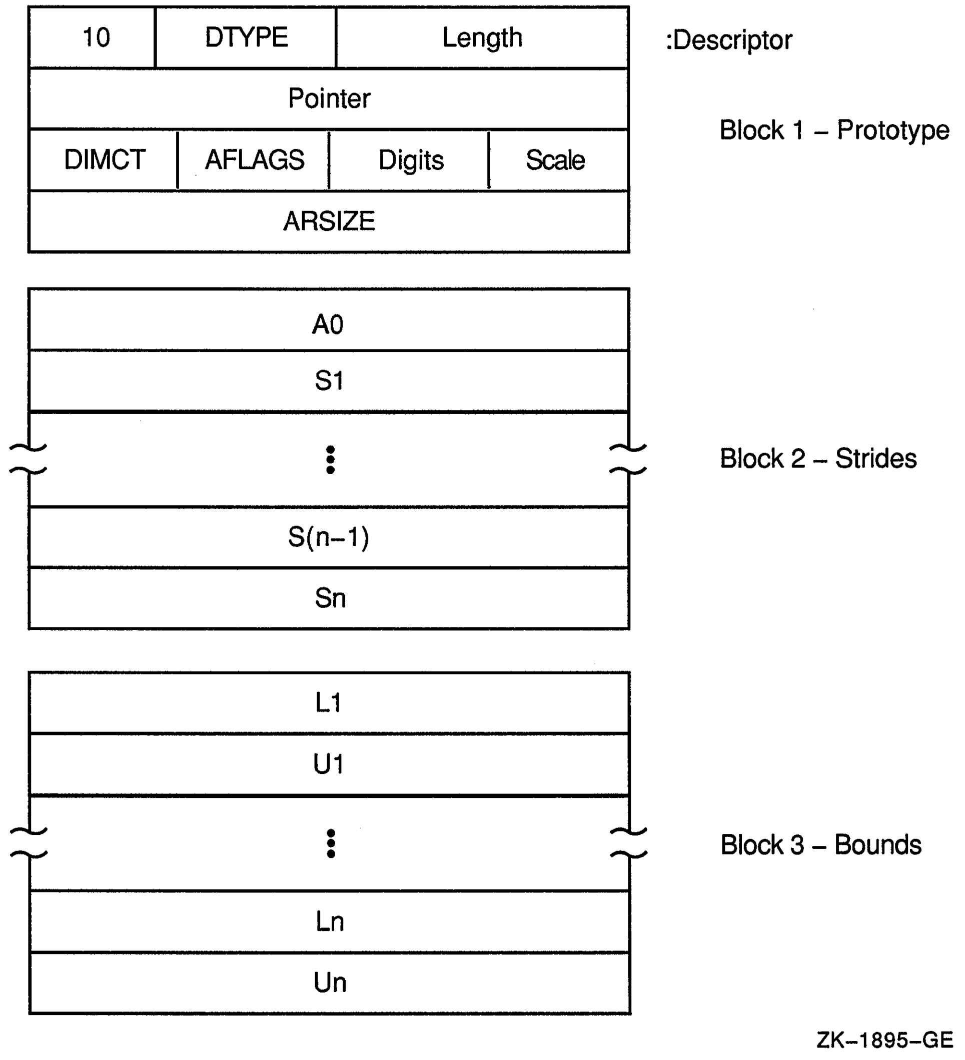
2.9.11 Noncontiguous Array Descriptor (DSC$K_CLASS_NCA)

The noncontiguous array descriptor describes an array in which the storage of the array elements can be allocated with a fixed, nonzero number of bytes separating logically adjacent elements. Two elements are said to be logically adjacent if their subscripts differ by 1 in the most rapidly varying dimension only. The difference between the addresses of two adjacent elements is termed the **stride**. Whether elements are stored by row or by column is the option of the calling program, and is automatically taken care of by a single accessing algorithm used by the called procedure.

This array descriptor is to be used where the calling program, at its option, can pass a slice of an array that contains noncontiguous allocations. This standard indicates no preference between the noncontiguous array descriptor (NCA) and the contiguous array descriptor (A), as described in Section 2.9.5, for language processors that always allocate contiguous arrays.

Figure 2–10 shows the format of a noncontiguous array descriptor, which consists of three contiguous blocks.

Figure 2–10 Noncontiguous Array Descriptor Format



Symbol	Description
DSC$W_LENGTH	Length of an array element in bytes, unless the DSC$B_DTYPE field contains the value *1* (aligned bit string) or *21* (packed-decimal string). Length of an array element is in bits for bit. Length of an array element is the number of 4-bit digits (not including the sign) for packed-decimal string.
DSC$B_DTYPE	A 1-byte data type code. Data type codes are listed in Sections 2.8.1 and 2.8.2.
DSC$B_CLASS	10 = DSC$K_CLASS_NCA.
DSC$A_POINTER	Address of first actual byte of data storage.

Symbol	Description	
DSC$B_SCALE <2,7:0>	Signed power-of-two or -ten multiplier, as specified by DSC$V_FL_BINSCALE, to convert the internal form to external form. (See Section 2.9.10.)	
DSC$B_DIGITS <2,15:8>	If nonzero, the unsigned number of decimal digits in the internal representation. If zero, the number of digits can be computed based on DSC$W_LENGTH. This field should be zero unless the DSC$B_TYPE field specifies a string data type that could contain numeric values.	
DSC$B_AFLAGS <2,23:16>	Array flag bits:	
	Reserved <2,18:16>	Reserved for future standardization by Digital. Must be zero.
	DSC$V_FL_BINSCALE <2,19>	If set, the scale factor specified by DSC$B_SCALE is a signed power-of-two multiplier to convert the internal form to external form. If not set, DSC$B_SCALE specifies a signed power-of-ten multiplier. (See Section 2.9.10.)
	DSC$V_FL_REDIM <2,20>	Must be zero.
	Reserved <2,23:21>	Reserved for future standardization by Digital. Must be zero.
DSC$B_DIMCT <2,31:24>	Number of dimensions, n.	
DSC$L_ARSIZE <3,31:0>	If the elements are actually contiguous, ARSIZE is the total size of the array (in bytes, unless the DSC$B_DTYPE field contains the value *21*—see description of DSC$W_LENGTH). If the elements are not allocated contiguously or if the program unit allocating the descriptor is uncertain whether the array is actually contiguous, the value placed in ARSIZE might be meaningless. For data type 1 (aligned bit string), DSC$W_LENGTH is in bits while DSC$L_ARSIZE is in bytes because the unit of length is in bits while the unit of allocation is aligned bytes.	
DSC$A_A0 <4,31:0>	Address of element A(0,0, . . . ,0). This need not be within the actual array. It is the same as DSC$A_POINTER for zero-origin arrays. DSC$A_A0 = POINTER - (S_1*L_1 + S_2*L_2 + . . . +S_n*L_n)	
DSC$L_Si <4+i,31:0>	Stride of the *i*th dimension. The difference between the addresses of successive elements of the *i*th dimension.	
DSC$L_Li <3+n+2*i,31:0>	Lower bound (signed) of the *i*th dimension.	
DSC$L_Ui <4+n+2*i,31:0>	Upper bound (signed) of the *i*th dimension.	

The following formulas specify the effective address, E, of an array element.

Warning: Modification of the following formulas is required if DSC$B_DTYPE equals 1 or 21 because DSC$W_LENGTH is given in bits or 4-bit digits rather than bytes.

The effective address, E, of A(I):

$$\begin{aligned} E &= A_0 + S_1*I \\ &= \text{POINTER} + S_1*[I - L_1] \end{aligned}$$

The effective address, E, of $A(I_1,I_2)$:

$$\begin{aligned} E &= A_0 + S_1*I_1 + S_2*I_2 \\ &= \text{POINTER} + S_1*[I_1 - L_1] + S_2*[I_2 - L_2] \end{aligned}$$

The effective address, E, of $A(I_1, . . . ,I_n)$:

$$\begin{aligned} E &= A_0 + S_1*I_1 + . . . + S_n*I_n \\ &= \text{POINTER} + S_1*[I_1 - L_1] + . . . + S_n*[I_n \\ &\quad - L_n] \end{aligned}$$

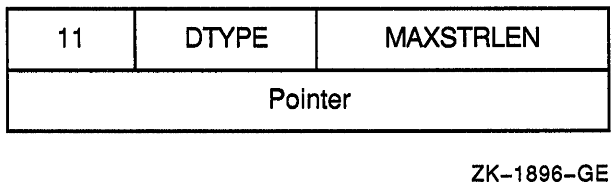
2.9.12 Varying String Descriptor (DSC$K_CLASS_VS)

A single descriptor form is used for varying string data types consisting of the following two fixed-length areas allocated contiguously with no padding in between:

CURLEN	An unsigned word specifying the current length in bytes of the immediately following string (byte aligned).
BODY	A fixed-length area containing the string that can vary from zero to a maximum length defined for each instance of string.

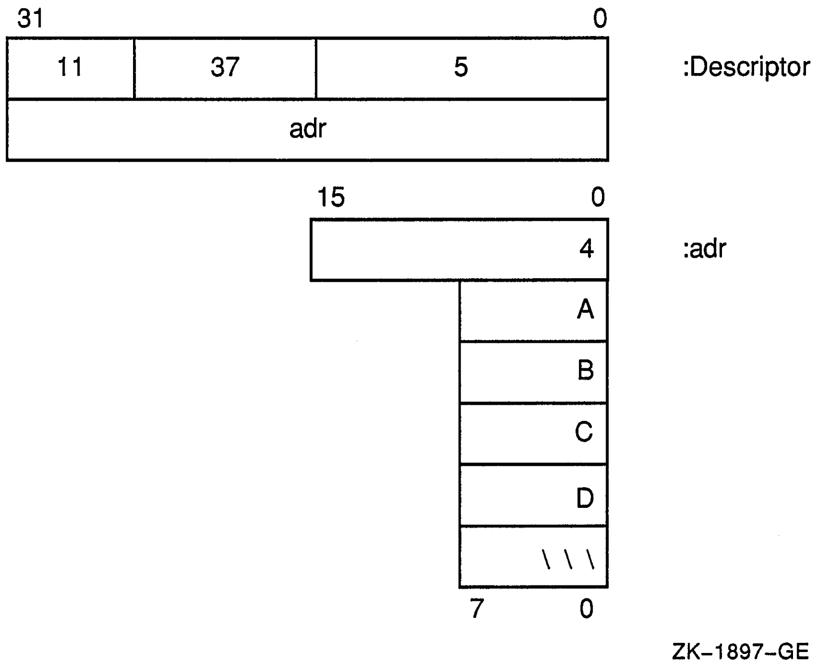
As an input parameter, this format is not interchangeable with class 1 (DSC$K_CLASS_S) or with class 2 (DSC$K_CLASS_D). When a called procedure modifies a varying string passed by reference or by descriptor, it writes the new length, *n*, into CURLEN and can modify all bytes of BODY. Figure 2–11 shows the format of a varying string descriptor.

Figure 2–11 Varying String Descriptor Format



Symbol	Description
DSC$W_MAXSTRLEN	Maximum length of the BODY field of the varying string in bytes in the range 0 to 2^{16}-1.
DSC$B_DTYPE	A 1-byte data type code that must have the value *37*, which specifies the varying character string data type (see Section 2.8.2 and Section 2.8.7). The use of other data types is reserved for future standardization.
DSC$B_CLASS	11 = DSC$K_CLASS_VS.
DSC$A_POINTER	Address of first field (CURLEN) of the varying string.

In the following example, MAXSTRLEN contains five, CURLEN contains four, string is currently ABCD, and the remaining byte is currently undefined.



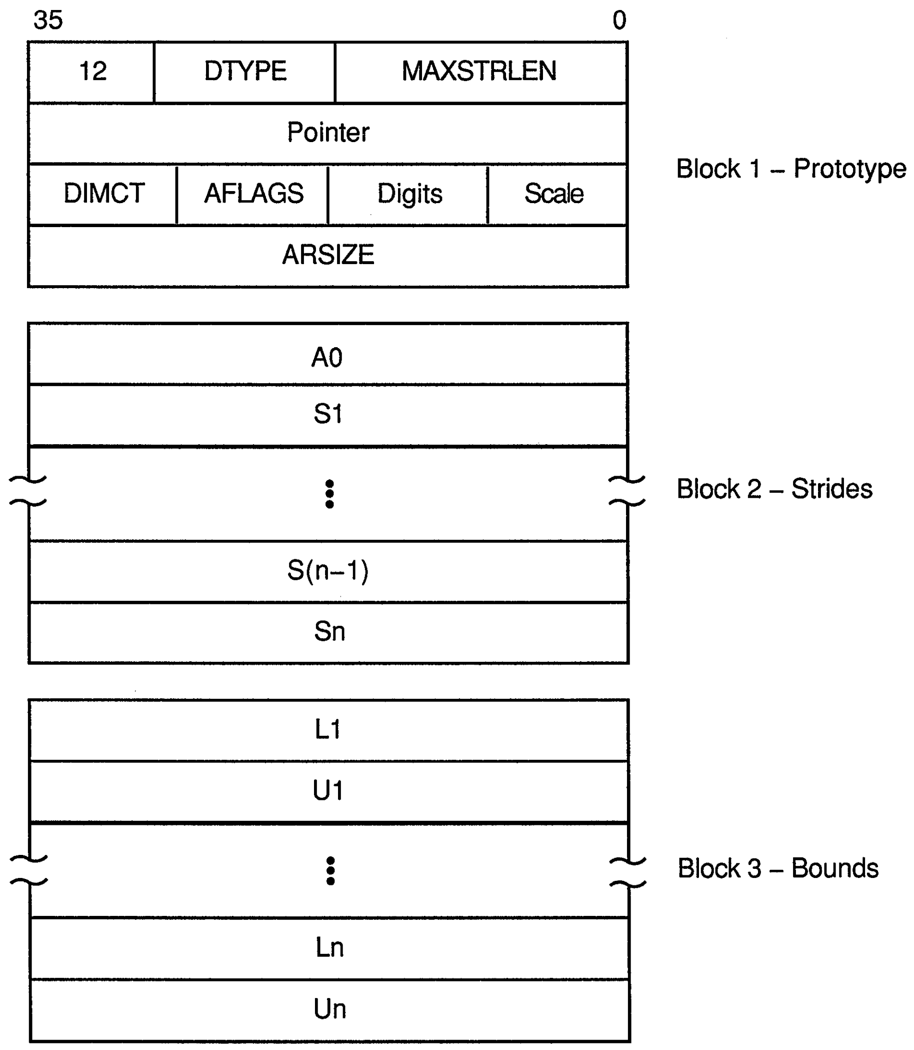
2.9.13 Varying String Array Descriptor (DSC$K_CLASS_VSA)

A variant of the noncontiguous array descriptor is used to specify an array of varying strings where each varying string has the same maximum length. Each array element is a varying string data type consisting of the following two fixed-length areas allocated contiguously with no padding in between:

CURLEN	An unsigned word specifying the current length in bytes of the immediately following string (byte aligned).
BODY	A fixed-length area containing the string that can vary from zero to the maximum length defined for an array element (MAXSTRLEN).

When a called procedure modifies a varying string in an array of varying strings passed to it by reference or by descriptor, it writes the new length, *n*, into CURLEN and can modify all bytes of BODY. The format of this descriptor is the same as the noncontiguous array descriptor except for the first two longwords. Figure 2–12 shows the format of a varying string array descriptor.

Figure 2–12 Varying String Array Descriptor Format



ZK–1898–GE

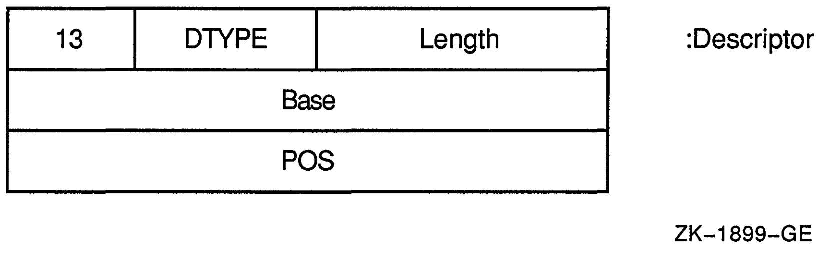
Symbol	Description
DSC$W_MAXSTRLEN	Maximum length of the BODY field of an array element in bytes in the range 0 to 2^{16}-1.
DSC$B_DTYPE	A 1-byte data type code that must have the value *37*, which specifies the varying character string data type (see Section 2.8.2 and Section 2.8.7). The use of other data types is reserved for future standardization.
DSC$B_CLASS	12 = DSC$K_CLASS_VSA.
DSC$A_POINTER	Address of first actual byte of data storage.

The remaining fields in the descriptor are identical to those in the noncontiguous array descriptor (NCA). The effective address computation of an array element produces the address of CURLEN of the desired element.

2.9.14 Unaligned Bit String Descriptor (DSC$K_CLASS_UBS)

A descriptor is used to pass an unaligned bit string (DSC$K_DTYPE_VU) that starts on an arbitrary bit boundary and ends on an arbitrary bit boundary. The length is 0 to 2^{16}-1 bits. The bit string can be accessed directly using the VAX variable bit field instructions. Therefore, the descriptor provides two components: a base address and a signed relative bit position. Figure 2–13 shows the format of an unaligned bit string descriptor.

Figure 2–13 Unaligned Bit String Descriptor Format



Symbol	Description
DSC$W_LENGTH	Length of data item in bits.
DSC$B_DTYPE	A 1-byte data type code that has the value *34*, which specifies the unaligned bit string data type (see 2.8.1 and 2.8.2). The use of other data types is reserved for future standardization.
DSC$B_CLASS	13 = DSC$K_CLASS_UBS

Symbol	Description
DSC$A_BASE	Base of address relative to which the signed relative bit position, POS, is used to locate the bit string. The base address need not be first actual byte of data storage.
DSC$L_POS	Signed longword relative bit position with respect to BASE of the first bit of unaligned bit string.

For example, a called procedure can use the following instruction to access a bit string of 32 bits or less:

```
EXTZV   DSC$L_POS(R0), DSC$W_LENGTH(R0), @DSC$A_BASE(R0), R1
```

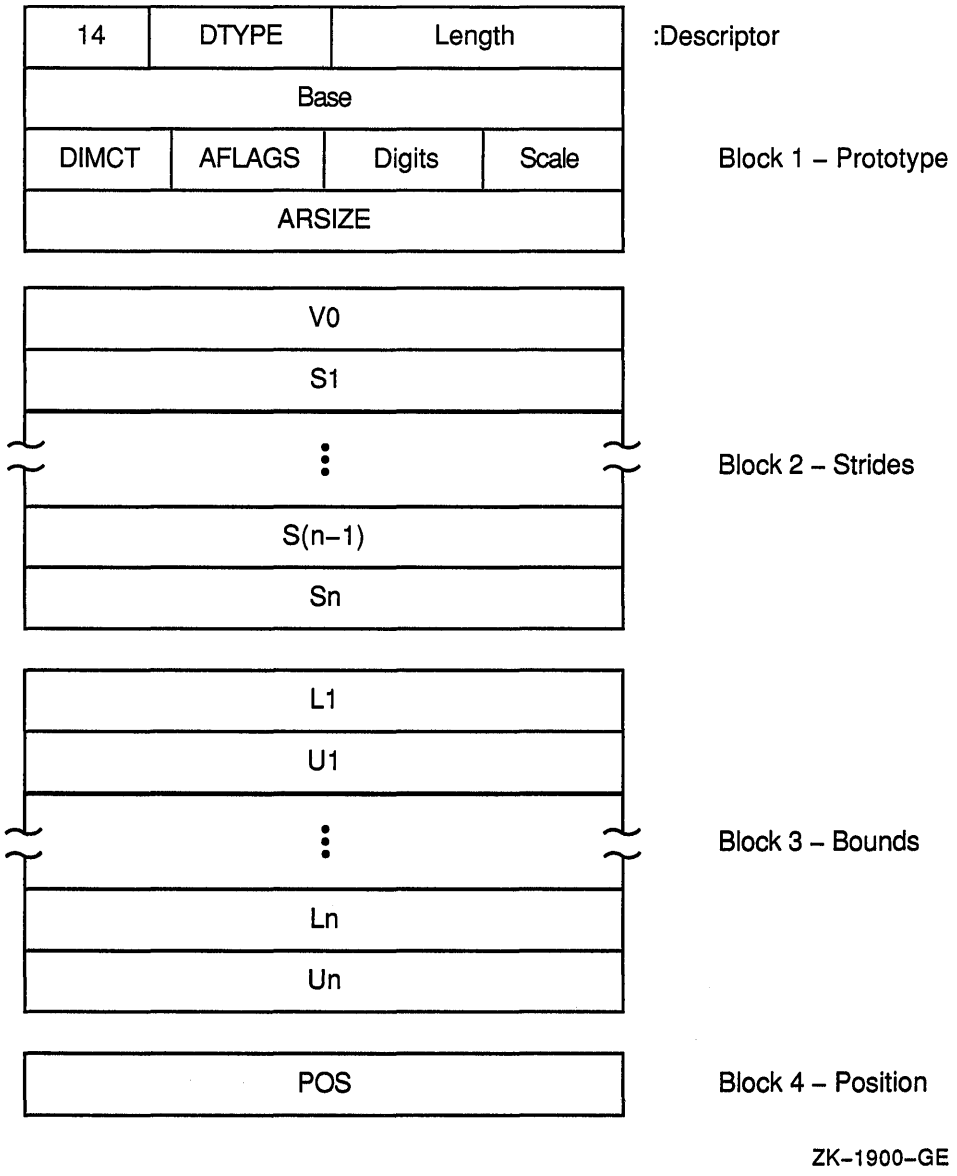
If R0 contains the address of the descriptor, this instruction copies the bit string to R1.

2.9.15 Unaligned Bit Array Descriptor (DSC$K_CLASS_UBA)

A variant of the noncontiguous array descriptor is used to specify an array of unaligned bit strings. Each array element is an unaligned bit string data type (DSC$K_DTYPE_VU) that starts on an arbitrary bit boundary and ends on an arbitrary bit boundary. The length of each element is the same and is 0 to 2^{16}-1 bits. You can access elements of the array directly, using the VAX variable bit field instructions. Therefore, the descriptor provides two components: a byte address, DSC$A_BASE, and a means to compute the signed bit offset, EB, with respect to BASE of an array element.

The unaligned bit array descriptor consists of four contiguous blocks that are always present. The first block contains the descriptor prototype information. Figure 2–14 shows the format of an unaligned bit array descriptor.

Figure 2–14 Unaligned Bit Array Descriptor Format



Symbol	Description	
DSC$W_LENGTH	Length of an array element in bits.	
DSC$B_DTYPE	A 1-byte data type code that must have the value *34*, which specifies the unaligned bit string data type (see Sections 2.8.1 and 2.8.2). The use of other data types is reserved for future standardization.	
DSC$B_CLASS	14 = DSC$K_CLASS_UBA.	
DSC$A_BASE	Base address relative to the effective bit offset, EB, used to locate elements of the array. The base address need not be the first actual byte of data storage.	
DSC$B_SCALE	Reserved for future standardization by Digital. Must be zero.	
DSC$B_DIGITS	If nonzero, the unsigned number of decimal digits in the internal representation. If zero, the number of digits can be computed based on DSC$W_LENGTH. This field should be zero unless the DSC$B_TYPE field specifies a string data type that could contain numeric values.	
DSC$B_AFLAGS <2,23:16>	Array flag bits:	
	Reserved <2,18:16>	Reserved for future standardization by Digital. Must be zero.
	DSC$V_FL_BINSCALE <2,19>	Must be zero.
	DSC$V_FL_REDIM <2,20>	Must be zero.
	Reserved <2,23:21>	Reserved for future standardization by Digital. Must be zero.
DSC$B_DIMCT <2,31:24>	Number of dimensions, n.	
DSC$L_ARSIZE <3,31:0>	If the elements are actually contiguous, ARSIZE is the total size of the array in bits. If the elements are not allocated contiguously or if the program unit allocating the descriptor is uncertain whether the array is actually contiguous, the value placed in ARSIZE might be meaningless.	
DSC$L_V0 <4,31:0>	Signed bit offset of element A(0, . . . ,0) with respect to BASE. $V_0 = POS - [S_1 * L_1 + \ldots + S_n * L_n]$.	
DSC$L_Si <4+i,31:0>	Stride of the *i*th dimension. The difference between the bit (not byte) addresses of successive elements of the *i*th dimension.	
DSC$L_Li <3+n+2*i,31:0>	Lower bound (signed) of the *i*th dimension.	
DSC$L_Ui <4+n+2*i,31:0>	Upper bound (signed) of the *i*th dimension.	
DSC$L_POS <5+n*3,31:0>	Signed longword relative bit position with respect to BASE of the first actual bit of the array, that is, element $A(L_1, \ldots, L_n)$.	

The signed, 32-bit effective bit offset, EB, of $A(I_1)$:

$$\begin{aligned} EB &= V_0 + S_1{*}I_1 \\ &= POS + S_1{*}[I_1 - L_1] \end{aligned}$$

The signed, 32-bit effective bit offset, EB, of $A(I_1,I_2)$:

$$\begin{aligned} EB &= V_0 + S_1{*}I_1 + S_2{*}I_2 \\ &= POS + S_1{*}[I_1 - L_1] + S_2{*}[I_2 - L_2] \end{aligned}$$

The signed, 32-bit effective bit offset, EB, of $A(I_1, \ldots, I_n)$:

$$\begin{aligned} EB &= V_0 + S_1{*}I_1 + \ldots + S_n{*}I_n \\ &= POS + S_1{*}[I_1 - L_1] + \ldots + S_n{*}[I_n \\ &\quad - L_n] \end{aligned}$$

Note that EB is computed ignoring integer overflow. EB is then usable as the position operand, and the content of DSC$A_BASE is usable as the base address operand in the VAX variable-length bit field instructions. Therefore, BASE must specify a byte that is within 2^{28} bytes of all bytes of storage in the bit array.

For example, consider a 1-origin, 1-dimension, 5-element array consisting of 3-bit elements allocated adjacently (therefore, S1 = 3). Assume BASE is byte 1000 and the first actual element, A(1), starts at bit <4> of byte 1001.

7	6	5	4	3	2	1	0	
								:1000
2	1	1	1			0		:1001
4	4	4	3	3	3	2	2	:1002
					5	5	5	:1003

ZK–1901–GE

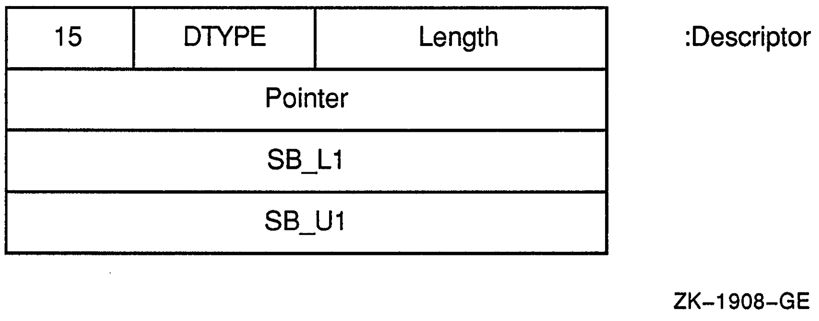
The following dependent field values occur in the descriptor:

$$\begin{aligned} POS &= 12 \\ V_0 &= 12 - 3{*}1 = 9 \end{aligned}$$

2.9.16 String with Bounds Descriptor (DSC$K_CLASS_SB)

A variant of the fixed-length string descriptor is used to specify strings where the string is viewed as a one-dimensional array with user-specified bounds. Figure 2–15 shows the format of a string with bounds descriptor.

Figure 2–15 String with Bounds Descriptor Format



Symbol	Description
DSC$W_LENGTH	Length of string in bytes.
DSC$B_DTYPE	A 1-byte data type code that must have the value *14*, which specifies the character string data type (see Sections 2.8.1 and 2.8.2). The use of other data types is reserved for future standardization.
DSC$B_CLASS	15 = DSC$K_CLASS_SB.
DSC$A_POINTER	Address of first byte of data storage.
DSC$L_SB_L1	Lower bound (signed) of the first (and only) dimension.
DSC$L_SB_U1	Upper bound (signed) of the first (and only) dimension.

The following formula specifies the effective address, E, of a string element A(I):

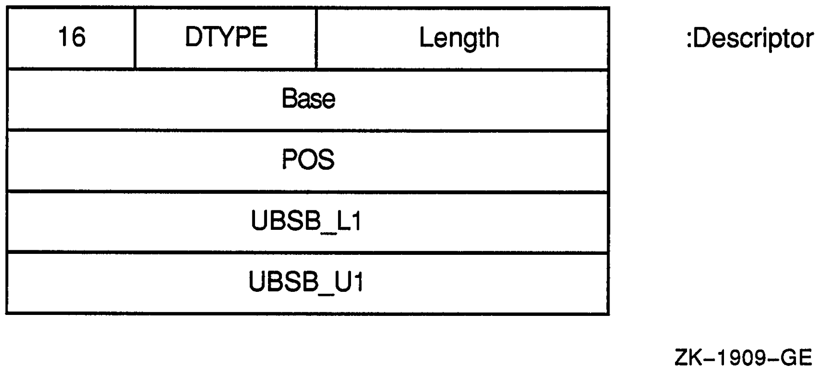
```
E = POINTER + [I - SB_L1]
```

If the string must be extended in a string comparison or assignment, the space character (hexadecimal 20 if ASCII) is used as the fill character.

2.9.17 Unaligned Bit String with Bounds Descriptor (DSC$K_CLASS_UBSB)

A variant of the unaligned bit string descriptor is used to specify bit strings where the string is viewed as a one-dimensional bit array with user-specified bounds. Figure 2–16 shows the format of an unaligned bit string with bounds descriptor.

Figure 2–16 Unaligned Bit String with Bounds Descriptor Format



Symbol	Description
DSC$W_LENGTH	Length of data item in bits.
DSC$B_DTYPE	A 1-byte data type code that must have the value *34*, which specifies the unaligned bit string data type (see Section 2.8.1 and Section 2.8.2). The use of other data types is reserved for future standardization.
DSC$B_CLASS	16 = DSC$K_CLASS_UBSB.
DSC$A_BASE	Base address relative to the signed relative bit position, POS, used to locate the bit string. The base address need not be the first actual byte of data storage.
DSC$L_POS	Signed longword relative bit position with respect to BASE of the first bit of the unaligned bit string.
DSC$L_UBSB_L1	Lower bound (signed) of the first (and only) dimension.
DSC$L_UBSB_U1	Upper bound (signed) of the first (and only) dimension.

The following formula specifies the effective bit offset, EB, of a bit element A(I):

```
EB = POS + [I - UBSB_L1]
```

2.9.18 Facility-Specific Descriptor Class Codes

Descriptor class codes 160 through 191 are reserved by Digital for facility-specific purposes. These codes must not be passed between facilities because different facilities might use the same code for different purposes. These codes can be used by compiler-generated code to pass parameters to the language-specific, run-time support procedures associated with that language or to VAX DEBUG.

2.9.19 Reserved Descriptor Class Codes

Descriptor classes 15 through 191 are reserved by Digital. Classes 192 through 255 are reserved for Digital's Computer Special Systems group and customers.

2.10 VAX Conditions

A condition is either (1) a hardware-generated synchronous exception or (2) a software event that is to be processed in a manner analogous to a hardware exception.

Floating-point overflow exception, memory access violation exception, and reserved operation exception are examples of hardware-generated conditions. An output conversion error, an end of file, or the filling of an output buffer are examples of software events that might be treated as conditions.

Depending on the condition and on the program, you can take four types of action when a condition occurs:

- Ignore the condition.

  For example, if an underflow occurs in a floating-point operation, continuing from the point of the exception with a zero result might be satisfactory.
- Take some special action and then continue from the point at which the condition occurred.

  For example, if the end of a buffer is reached while a series of data items are being written, the special action is to start a new buffer.
- End the operation and branch from the sequential flow of control.

  For example, if the end of an input file is reached, the branch exits from a loop that is processing the input data.
- Treat the condition as an unrecoverable error.

  For example, when the floating divide-by-zero exception condition occurs, the program exits after writing (optionally) an appropriate error message.

When an unusual event or error occurs in a called procedure, the procedure can return a condition value to the caller indicating what has happened (see Section 2.5). The caller tests the condition value and takes the appropriate action.

When an exception is generated by the hardware, a branch out of the program's flow of control occurs automatically. In this case, and for certain software-generated events, it is more convenient to handle the condition as soon as it is detected rather than to program explicit tests.

2.10.1 Condition Handlers

To handle hardware- or software-detected exceptions, the VAX Condition Handling Facility allows you to specify a condition handler procedure to be called when an exception condition occurs.

An active procedure can establish a condition handler to be associated with it. The presence of a condition handler is indicated by a nonzero address in the first longword of the procedure's stack frame. When an event occurs that is to be treated using the condition-handling facility, the procedure detecting the event signals the event by calling the facility and passing a condition value describing the condition that occurred. This condition value has the format and interpretation as described in Section 2.5. All hardware exceptions are signaled.

When a condition is signaled, the condition-handling facility looks for a condition handler in the current procedure's stack frame. If a handler is found, it is entered. If no handler is associated with the current procedure, the immediately preceding stack frame is examined. Again, if a handler is found, it is entered. If a handler is not found, the search of previous stack frames continues until the default condition handler established by the system is reached or the stack runs out.

The default condition handler prints messages indicated by the signal argument list by calling the Put Message (SYS$PUTMSG) system service, followed by an optional symbolic stack traceback. Success conditions with STS$K_SUCCESS result in messages to SYS$OUTPUT only. All other conditions, including informational messages (STS$K_INFO), produce messages on SYS$OUTPUT and SYS$ERROR.

For example, if a procedure needs to keep track of the occurrence of the floating-point underflow exception, it can establish a condition handler to examine the condition value passed when the handler is invoked. Then, when the floating-point underflow exception occurs, the condition handler is entered and logs the condition. The handler returns to the instruction immediately following the instruction causing the underflow.

If floating-point operations occur in many procedures of a program, the condition handler can be associated with the program's main procedure. When the condition is signaled, successive stack frames are searched until the stack frame for the main procedure is found, at which time the handler is entered. If a user program has not associated a condition handler with any of the procedures that are active at the time of the signal, successive stack frames are searched until the frame for the system program invoking the user program is reached. A default condition handler that prints an error message is then entered.

2.10.2 Condition Handler Options

Each procedure activation potentially has a single condition handler associated with it. This condition handler is entered whenever any condition is signaled within that procedure. (It can also be entered as a result of signals within active procedures called by the procedure.) Each signal includes a condition value (see Section 2.5) that describes the condition causing the signal. When the condition handler is entered, the condition value should be examined to determine the cause of the signal. After the handler processes the condition or ignores it, it can take one of the following actions:

- Return to the instruction immediately following the signal. Note that it is not always possible to make such a return.
- Resignal the same or a modified condition value. A new search for a condition handler begins with the immediately preceding stack frame.
- Signal a different condition.
- Unwind the stack.

2.11 Operations Involving Condition Handlers

The VAX Condition Handling Facility provides functions to perform the following operations:

- Establish a condition handler.

  A condition handler is associated with the current procedure by placing the handler's address in the current procedure's activation stack frame.
- Revert to the caller's handling.

  If a condition handler has been established, you can remove it by clearing its address in the current procedure activation's stack frame.
- Enable or disable certain arithmetic exceptions.

  The software can enable or disable the following hardware exceptions: floating-point underflow, integer overflow, and decimal overflow. No signal occurs when the exception is disabled.
- Signal a condition.

  Signaling a condition initiates the search for an established condition handler.
- Unwind the stack.

  Upon exit from a condition handler it is possible to remove one or more frames occurring before the signal from the stack. During the unwinding operation, the stack is scanned; if a condition handler is associated with a frame, that handler is entered before the frame is removed. Unwinding the stack allows a procedure to perform application-specific cleanup operations before exiting.

2.11.1 Establishing a Condition Handler

Each procedure activation has a condition handler potentially associated with it, using longword 0 in its stack frame. Initially, longword 0 contains the value *0*, indicating no handler. You establish a handler by moving the address of the handler's procedure entry point mask to the establisher's stack frame.

In addition, the VMS operating system provides three statically allocated exception vectors for each access mode of a process. These vectors are available to declare condition handlers that take precedence over any handlers declared at the procedure level. These are used, for example, to allow a debugger to monitor all exceptions and for the system to establish a last-chance handler. Because these handlers do not obey the procedure-nesting rules, they should not be used by modular code. Instead, the stack-based declaration should be used.

The following code establishes a condition handler:

```
MOVAB handler_entry_point,0(FP)
```

2.11.2 Reverting to the Caller's Handling

Reverting to the caller's handling deletes the condition handler associated with the current procedure activation. You do this by clearing the handler address in the stack frame.

The code to revert to the caller's handling is as follows:

```
CLRL 0(FP)
```

2.11.3 Signaling a Condition

The signal operation is the method used for indicating the occurrence of an exception condition. To issue a message and be able to continue execution after handling the condition, a program calls the LIB$SIGNAL procedure, as follows:

```
CALL LIB$SIGNAL (condition-value, arg_list...)
```

To issue a message, but not continue execution, a program calls LIB$STOP, as follows:

```
CALL LIB$STOP (condition-value, arg_list...)
```

In both cases, the **condition-value** argument indicates the condition that is signaled. However, LIB$STOP sets the severity of the **condition-value** argument to be a severe error. The remaining arguments describe the details of the exception. These are the same arguments used to issue a system message.

Note that, unlike most calls, LIB$SIGNAL and LIB$STOP preserve R0 and R1 as well as the other registers. Therefore, a debugger can insert a call to LIB$SIGNAL to display the entire state of the process at the time of the exception. It also allows signals to be coded in VAX MACRO without changing the register usage. This feature of preserving R0 and R1 is useful for debugging checks and gathering statistics. Hardware and system service exceptions behave like calls to LIB$SIGNAL.

The signal procedure examines the two exception vectors first, then up to a system-defined maximum number of previous stack frames, and finally the last-chance exception vector, if necessary. The current and previous stack frames are found by using FP and chaining back through the stack frames using the saved FP in each frame. The exception vectors have three address locations per access mode.

As part of image startup, the system declares a default last-chance handler. This handler is used as a last resort when the normal handlers are not performing correctly. The debugger can replace the default system last-chance handler with its own.

In some frame before the call to the main program, the system establishes a default catch-all condition handler that issues system messages. In a subsequent frame before the call to the main program, the system usually establishes a traceback handler. These system-supplied condition handlers use the **condition-value** argument to get the message and then use the remainder of the argument list to format and output the message through the system service, SYS$PUTMSG.

If the severity field of the **condition-value** argument (bits <2:0>) does not indicate a severe error (that is, a value of *4*), these default condition handlers return with SS$_CONTINUE. If the severity is a severe error, these default handlers exit the program image with the condition value as the final image status.

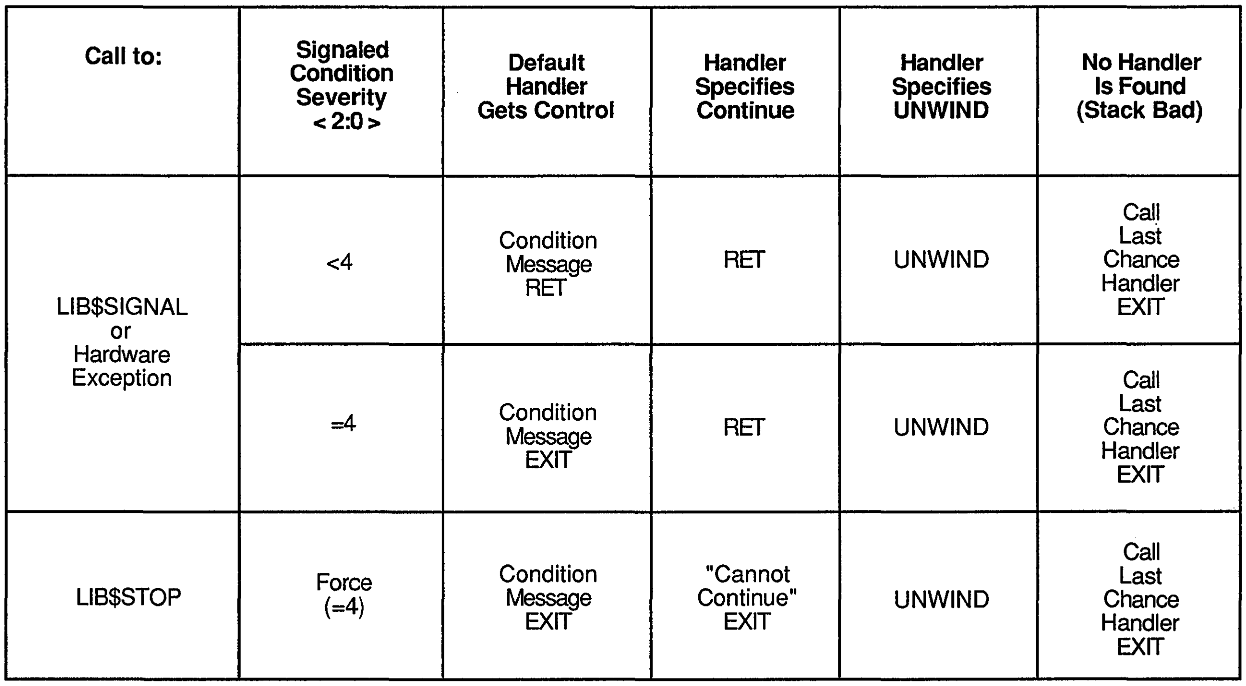
The stack search ends when the old FP is *0* or is not accessible, or when a system-defined maximum number of frames have been examined. If no condition handler is found, or all handlers returned with a SS$_RESIGNAL, then the vectored last-chance handler is called.

If a handler returns SS$_CONTINUE, and LIB$STOP was not called, control returns to the signaler. Otherwise, LIB$STOP issues a message indicating that an attempt was made to continue from a noncontinuable exception and exits with the condition value as the final image status.

Figure 2–17 lists all combinations of interaction between condition handler actions, the default condition handlers, the types of signal, and the call to signal or stop. In the table, "cannot continue" indicates an error that results in the following message:

```
IMPROPERLY HANDLED CONDITION, ATTEMPT TO CONTINUE FROM STOP.
```

Figure 2–17 Interaction Between Handlers and Default Handlers



Call to:	Signaled Condition Severity < 2:0 >	Default Handler Gets Control	Handler Specifies Continue	Handler Specifies UNWIND	No Handler Is Found (Stack Bad)
LIB$SIGNAL or Hardware Exception	<4	Condition Message RET	RET	UNWIND	Call Last Chance Handler EXIT
	=4	Condition Message EXIT	RET	UNWIND	Call Last Chance Handler EXIT
LIB$STOP	Force (=4)	Condition Message EXIT	"Cannot Continue" EXIT	UNWIND	Call Last Chance Handler EXIT

ZK–4247–GE

2.12 Properties of Condition Handlers

The following subsections describe the properties of condition handlers.

2.12.1 Condition Handler Parameters and Invocation

If a condition handler is found on a software-detected exception, the handler is called with the following argument list:

```
continue = handler (signal_args, mechanism_args)
```

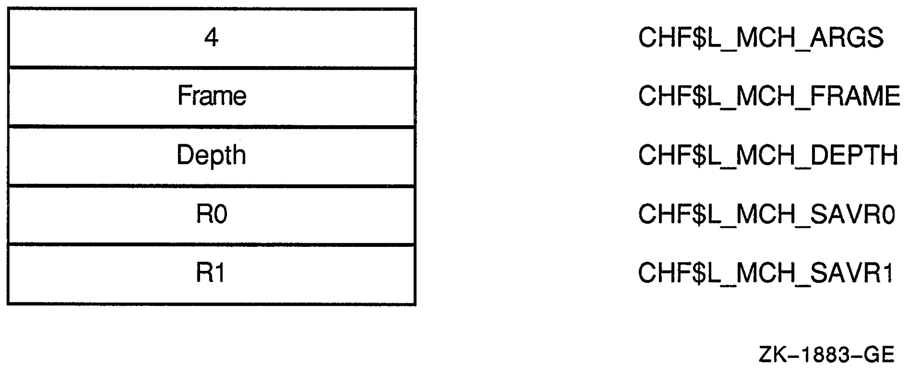
Each argument is a reference to a longword vector. The first longword of each vector is the number of remaining longwords in the vector. You can use the symbols CHF$L_SIGARGLST (=4) and CHF$L_MCHARGLST (=8) to access the condition handler arguments relative to AP.

The **signal_args** longword is the condition argument list from the call to LIB$SIGNAL or LIB$STOP, expanded to include the PC and PSL of the next instruction to execute on a continue. The second longword is the condition value being signaled.

Because bits <2:0> of the condition value indicate severity and do not indicate which condition is being signaled, the handler should examine only the condition identification, that is, condition value bits <27:3>. The setting of bits <2:0> varies depending upon the environment. In fact, some handlers might only change the severity of a condition and resignal. The symbols CHF$L_SIG_ARGS (=0) and CHF$L_SIG_NAME (=4) can be used to refer to the elements of the signal vectors.

Figure 2–18 shows the format of the mechanism argument vector.

Figure 2–18 Format of the Mechanism Argument Vector



The frame is the contents of the FP in the establisher's context. If the restrictions described in Section 2.12.3 are met, the frame can be used as a base to access the local storage of the establisher.

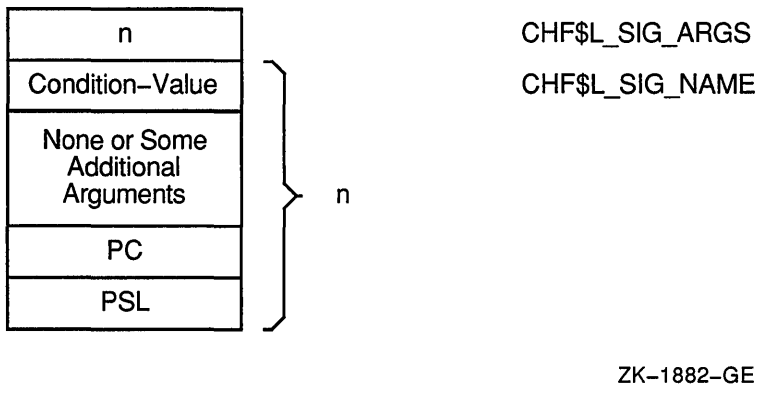
The depth is a positive count of the number of procedure activation stack frames between the frame in which the exception occurred and the frame depth that established the handler being called. Depth has the value *0* for an exception handled by the procedure activation invoking the exception (that is, containing the instruction causing the hardware exception or calling LIB$SIGNAL). Depth has positive values for procedure activations calling the one having the exception, for example, 1 for the immediate caller.

If a system service gives an exception, the immediate caller of the service is notified at depth = *1*. Depth has value *–2* when the condition handler is established by the primary exception vector, *–1* when it is established by the secondary vector, and *–3* when it is established by the last-chance vector.

The contents of R0 and R1 are the same as at the time of the call to LIB$SIGNAL or LIB$STOP.

For hardware-detected exceptions, the condition value indicates which exception vector was taken; the next 0 or several longwords are additional parameters. The remaining two longwords are the PC and PSL.
Figure 2–19 shows the format of the signal argument vector.

Figure 2–19 Format of the Signal Argument Vector



If the VAX vector hardware or emulator option is in use, then for hardware detected exceptions, the vector state is implicitly saved before any condition handler is entered and restored after the condition handler returns. (No save/restore is required for exceptions that are initiated by calls to LIB$SIGNAL or LIB$STOP because there can be no useful vector state at the time of such calls in accordance with the rules for Vector Register Usage in Section 2.6.2.) A condition handler can thus make use of the system vector facilities in the same manner as mainline code.

The saved vector state is not directly available to a condition handler. A condition handler that needs to manipulate the vector state to carry out agreements with its callers can call the SYS$RESTORE_VP_STATE service. This service restores the saved state to the vector registers (whether hardware or emulated) and cancels any subsequent restore. The vector state can then be manipulated directly in the normal manner by means of vector instructions. (This service is normally of interest only during processing for an unwind condition.)

2.12.2 System Default Condition Handlers

If one of the default condition handlers established by the system is entered, it calls the system service SYS$PUTMSG to interpret the signal argument list and output the indicated information or error message. See the description of SYS$PUTMSG in the *VMS System Services Reference Manual* for the format of the signal argument list.

2.12.3 Use of Memory

A condition handler and the procedures it calls can refer only to explicitly passed arguments. Handlers cannot refer to COMMON or other external storage, and they cannot reference local storage in the procedure that established the handler. The existence of handlers does not affect compiler optimization. Compilers that do not follow this rule must ensure that any variables referred to by the handler are always in memory.

2.12.4 Returning from a Condition Handler

Condition handlers are invoked by the VAX Condition Handling Facility. Therefore, the return from the condition handler is to the condition-handling facility.

To continue from the instruction following the signal, the handler must return with the function value SS$_CONTINUE (*true*), that is, with bit <0> set. If, however, the condition is signaled with a call to LIB$STOP, the image exits. To resignal the condition, the condition handler returns with the function value SS$_RESIGNAL (*false*), that is, with bit <0> clear. To alter the severity of the signal, the handler modifies the low-order three bits of the condition value longword in the **signal-args** vector and resignals. If the condition handler wants to alter the defined control bits of the signal, the handler modifies bits <31:28> of the condition value and resignals. To unwind, the handler calls SYS$UNWIND and then returns. In this case, the handler function value is ignored.

2.12.5 Request to Unwind

To unwind, the handler or any procedure it calls can make the following call:

```
success = SYS$UNWIND
            ( [depadr =  handler depth  + 1],
              [new_PC =  return PC ] )
```

The argument **depadr** specifies the address of the longword containing the number of presignal frames (depth) to be removed. If that number is less than or equal to *0*, then nothing is to be unwound. The default (address=0) is to return to the caller of the procedure that established the handler that issued the $UNWIND service. To unwind to the establisher, the depth from the call to the handler should be specified. When the handler is at depth 0, it can achieve the equivalent of an unwind operation to an arbitrary place in its establisher by altering the PC in its **signal-args** vector and returning with SS$_CONTINUE instead of performing an unwind.

The argument **new_PC** specifies the location to receive control when the unwinding operation is complete. The default is to continue at the instruction following the call to the last procedure activation removed from the stack.

The function value SUCCESS is either a standard success code (SS$_NORMAL), or it indicates failure with one of the following return status condition values:

- No signal active (SS$_NOSIGNAL)
- Already unwinding (SS$_UNWINDING)
- Insufficient frames for depth (SS$_INSFRAME)

The unwinding operation occurs when the handler returns to the condition- handling facility. Unwinding is done by scanning back through the stack and calling each handler that has been associated with a frame. The handler is called with exception SS$_UNWIND to perform any application-specific cleanup. If the depth specified includes unwinding the establisher's frame, the current handler is recalled with this unwind exception.

The call to the handler takes the same form as previously described, with the following values:

```
signal_args
        1
        condition_value = SS$_UNWIND

mechanism_args
        4
        frame   establisher's frame
        depth   0 (that is, unwinding self)
        R0      R0 that unwind will restore
        R1      R1 that unwind will restore
```

After each handler is called, the stack is cut back to the previous frame.

Note that the exception vectors are not checked because they are not being removed. Any function value from the handler is ignored. To specify the value of the top-level function being unwound, the handler should modify R0 and R1 in the **mechanism_args** vector. They are restored from the **mechanism_args** vector at the end of the unwind. Depending on the arguments to SYS$UNWIND, the unwinding operation is terminated as follows:

SYS$UNWIND(0,0)	Unwind to the establisher's caller with the establisher function value restored from R0 and R1 in the **mechanism-args** vector.
SYS$UNWIND(depth,0)	Unwind to the establisher at the point of the call that resulted in the exception. The contents of R0 and R1 are restored from R0 and R1 in the **mechanism_args** vector.
SYS$UNWIND(depth,location)	Unwind to the specified procedure activation and transfer to a specified location with the contents of R0 and R1 from R0 and R1 in the **mechanism_args** vector.

You can call SYS$UNWIND whether the condition was a software exception signaled by calling LIB$SIGNAL or LIB$STOP or was a hardware exception. Calling SYS$UNWIND is the only way to continue execution after a call to LIB$STOP.

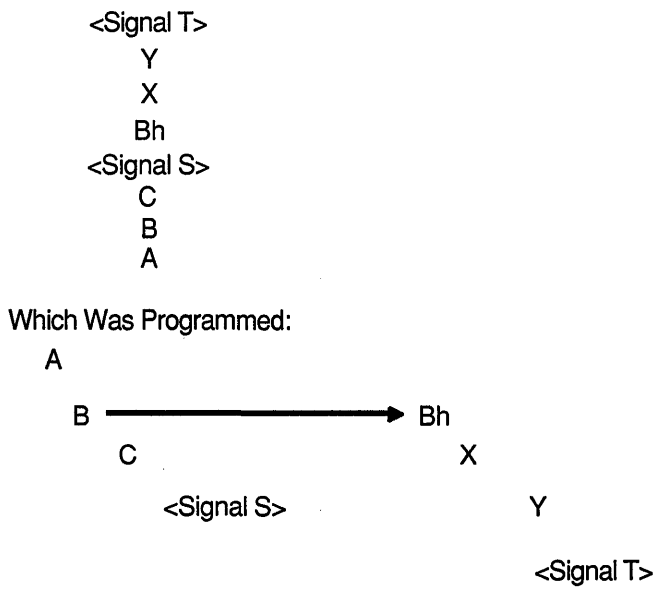
2.12.6 Signaler's Registers

Because the handler is called and can in turn call routines, the actual register values in use at the time of the signal or exception can be scattered on the stack. To find the registers R2 through FP, a scan of stack frames must be performed starting with the current frame and ending with the call to the handler. During the scan, the last frame found to save a register contains that register's contents at the time of the exception. If no frame saved the register, the register is still active in the current procedure. The frame of the call to the handler can be identified by the return address of SYS$CALL_HANDL+4. Thus, the registers are as follows:

R0, R1	In mechanism_args
R2..R11	Last frame saving it
AP	Old AP of SYS$CALL_HANDL+4 frame
FP	Old FP of SYS$CALL_HANDL+4 frame
SP	Equal to end of signal-args vector+4
PC, PSL	At end of signal-args vector

2.13 Multiple Active Signals

A signal is said to be active until the signaler gets control again or is unwound. A signal can occur while a condition handler or a procedure it has called is executing in response to a previous signal. For example, procedures A, B, and C establish condition handlers Ah, Bh, and Ch. If A calls B and B calls C, which signals S, and Ch resignals, then Bh gets control. If Bh calls procedure X, and X calls procedure Y, and Y signals T, the stack is as follows:



ZK-1884-GE

The handlers are searched for in the following order: Yh, Xh, Bhh, Ah. Note that Ch is not called because it is a structural descendant of B. Bh is not called again because that would require it to be recursive. Recursive handlers cannot be coded in nonrecursive languages such as FORTRAN. Instead, Bh can establish itself or another procedure as its handler (Bhh).

The following algorithm is used on the second and subsequent signals that occur before the handler for the original signal returns to the condition-handling facility. The primary and secondary exception vectors are checked. Then, however, the search backward in the process stack is modified. In effect, the stack frames traversed in the first search are skipped over in the second search. Thus, the stack frame preceding the first condition handler, up to and including the frame of the procedure that has established the handler, is skipped. Despite this skipping, depth is not incremented. For example, the stack frames traversed in the first and second search are skipped over in a third search. Note that if a condition handler signals, it is not automatically invoked recursively. However, if a handler itself establishes a handler, this second handler will be invoked. Thus, a recursive condition handler should start by establishing itself. Any procedures invoked by the handler are treated in the normal way; that is, exception signaling follows the stack up to the condition handler.

If an unwinding operation is requested while multiple signals are active, all the intermediate handlers are called for the operation. For example, in the preceding diagram, if Ah specifies unwinding to A, the following handlers are called for the unwind: Yh, Xh, Bhh, Ch, and Bh.

For proper hierarchical operation, an exception that occurs during execution of a condition handler established in an exception vector should be handled by that handler rather than propagating up the activation stack. To prevent such propagation, the vectored condition handler should establish a handler in its stack frame to handle all exceptions.

A VMS Data Types

This appendix describes the structures of the VMS operating system data types and ones that support the higher level languages.

A.1 VMS Data Types

The VMS Usage entry in the documentation format for system routines indicates the VMS data type of the argument. Each VMS data type has only one storage representation. For example, the VMS data type **access_mode** is an unsigned byte. In addition, a VMS data type may or may not have a conceptual meaning.

Most VMS data types may be considered conceptual types; that is, they carry meaning unique in the context of the VMS operating system. The **access_mode** is one of these. The storage representation of this VMS type is an unsigned byte, and the conceptual content of this unsigned byte is the fact that it designates a hardware access mode and therefore has only four valid values: *0*, designating kernel mode; *1*, executive mode; *2*, supervisor mode; and *3*, user mode. However, some VMS data types are not conceptual types; that is, they specify a storage representation, but carry no other semantic content from the point of view of VMS. For example, the VMS data type **byte_signed** is not a conceptual type.

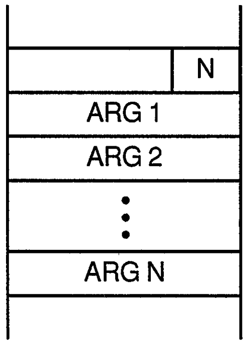
Note: The VMS Usage entry is NOT a traditional data type such as the VAX standard data types—byte, word, longword, and so on. It is significant only within the VMS operating system environment and is intended solely to expedite data declarations within application programs.

To use the VMS Usage entry, perform the following procedure:

1. Find the data type in Table A–1 and read its definition.
2. Find the same VMS data type in the appropriate VAX language implementation table (Tables A–2 through A–13) and its corresponding source language type declaration.
3. Use this code as your type declaration in your application program. Note that, in some instances, you may have to modify the declaration.

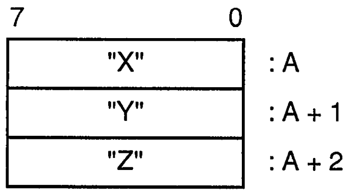
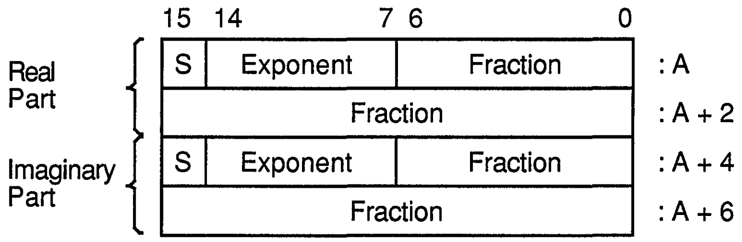
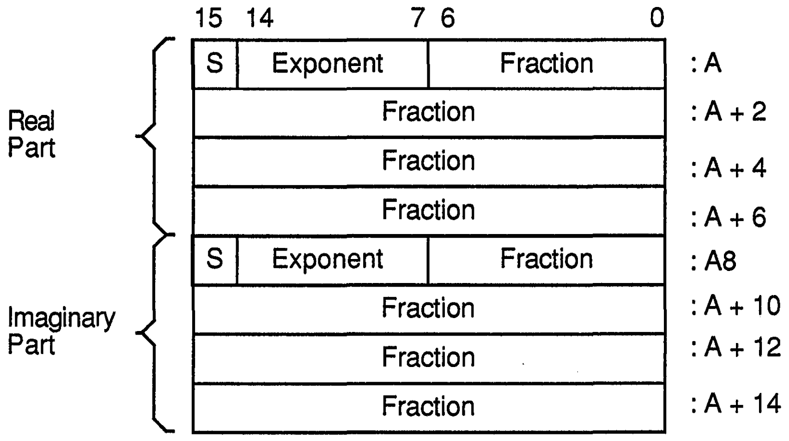
Table A–1 lists and describes the VMS data types.

Table A–1 VMS Data Types

Data Type	Definition
access_bit_names	Homogeneous array of 32 quadword descriptors; each descriptor defines the name of one of the 32 bits in an access mask. The first descriptor names bit <0>, the second descriptor names bit <1>, and so on.
access_mode	Unsigned byte denoting a hardware access mode. This unsigned byte can take four values: *0* specifies kernel mode; *1*, executive mode; *2*, supervisor mode; and *3*, user mode.
address	Unsigned longword denoting the virtual memory address of either data or code, but not of a procedure entry mask (which is of type **procedure**).
address_range	Unsigned quadword denoting a range of virtual addresses, which identify an area of memory. The first longword specifies the beginning address in the range; the second longword specifies the ending address in the range.
arg_list	Procedure argument list consisting of 1 to 256 longwords. The first longword contains an unsigned integer count of the number of successive, contiguous longwords, each of which is an argument to be passed to a procedure by means of a VAX CALL instruction. The argument list has the following format:  ZK–4204–GE
ast_procedure	Unsigned longword integer denoting the entry mask to a procedure to be called at AST level. (Procedures that are not to be called at AST level are of type **procedure.**)
boolean	Unsigned longword denoting a Boolean truth value flag. This longword can have only two values: *1* (true) and *0* (false).
byte_signed	Is the same as the data type **byte integer (signed)** in Table 1–3.
byte_unsigned	Is the same as the data type **byte (unsigned)** in Table 1–3.
channel	Unsigned word integer that is an index to an I/O channel.
char_string	String of from 0 to 65,535 8-bit characters. This VMS data type is the same as the data type **character string** in Table 1–3. The following diagram shows the character string XYZ:

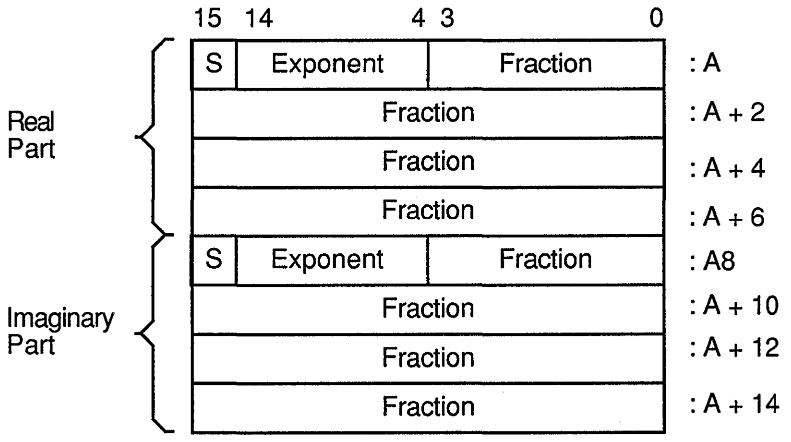
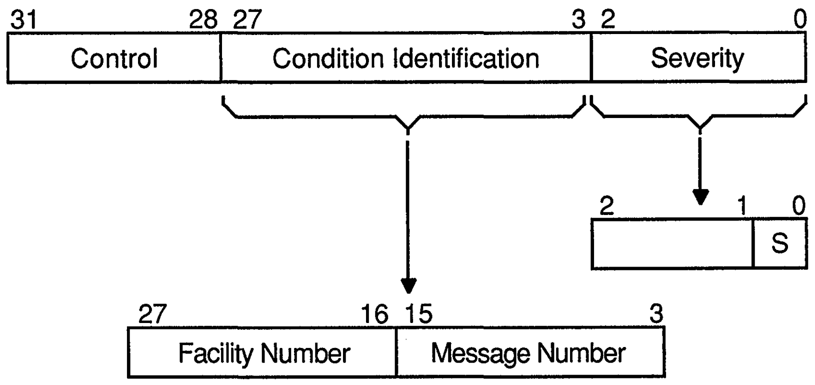
(continued on next page)

Table A–1 (Cont.) VMS Data Types

Data Type	Definition
	 ZK–4202–GE
complex_number	One of the VAX standard complex floating-point data types. The three complex floating-point numbers are F_floating complex, D_floating complex, and G_floating complex. An F_floating complex number (r,i) is comprised of two F_floating point numbers: the first is the real part (r) of the complex number; the second is the imaginary part (i). The structure of an F_floating complex number is as follows:  15 14 7 6 0 Real Part Imaginary Part S Exponent Fraction Fraction S Exponent Fraction Fraction :A :A + 2 :A + 4 :A + 6 ZK–4203–GE A D_floating complex number (r,i) is comprised of two D_floating point numbers: the first is the real part (r) of the complex number; the second is the imaginary part (i). The structure of a D_floating complex number is as follows:  15 14 7 6 0 Real Part Imaginary Part S Exponent Fraction Fraction Fraction Fraction S Exponent Fraction Fraction Fraction Fraction :A :A + 2 :A + 4 :A + 6 :A8 :A + 10 :A + 12 :A + 14 ZK–4201–GE

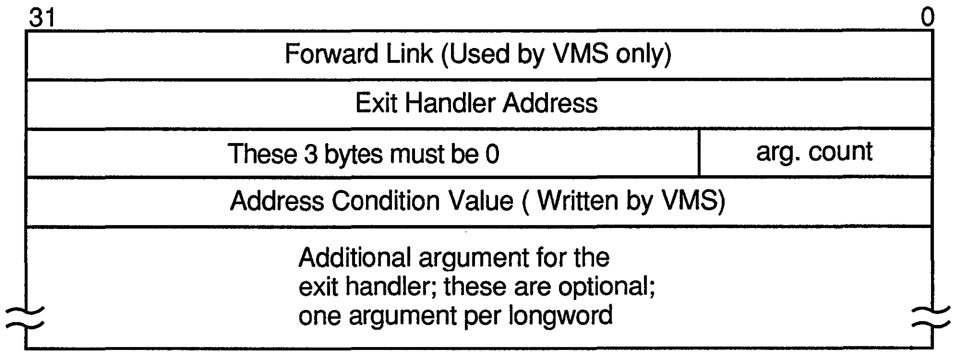
(continued on next page)

Table A–1 (Cont.) VMS Data Types

Data Type	Definition
	A G_floating complex number (r,i) is comprised of two G_floating point numbers: the first is the real part (r) of the complex number; the second is the imaginary part (i). The structure of a G_floating complex number is as follows:  15 14 4 3 0 S Exponent Fraction :A :A + 2 :A + 4 :A + 6 : A8 :A + 10 :A + 12 :A + 14 Real Part Imaginary Part ZK–4200–GE
cond_value	Unsigned longword integer denoting a condition value (that is, a return status or system condition code) that is typically returned by a procedure in R0. Each numeric condition value has a unique symbolic name in the following format, where code is a mnemonic describing the return condition:  31 28 27 3 2 0 Control Condition Identification Severity 2 1 0 S 27 16 15 3 Facility Number Message Number ZK–1795–GE

(continued on next page)

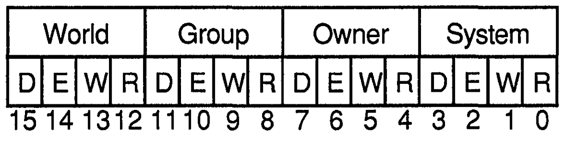
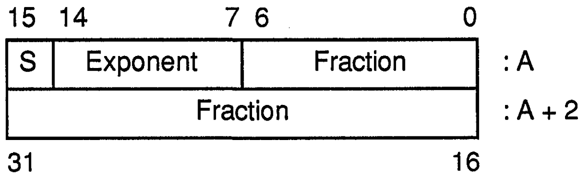
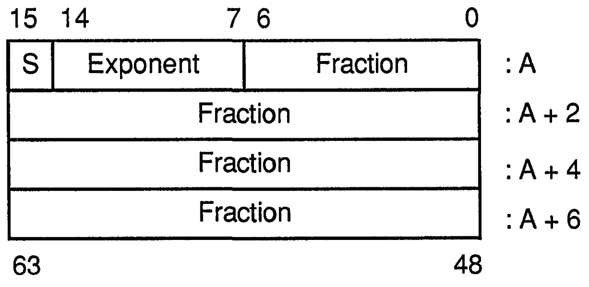
Table A–1 (Cont.) VMS Data Types

Data Type	Definition
	Depending on your specific needs, you can test just the low-order bit, the low-order three bits, or the entire value. • The low-order bit indicates successful (1) or unsuccessful (0) completion of the service. • The low-order three bits taken together represent the severity of the error. • The remaining bits <31:3> classify the particular return condition and the operating system component that issued the condition value.
context	Unsigned longword used by a called procedure to maintain position over an iterative sequence of calls. It is usually initialized by the caller, but thereafter manipulated by the called procedure.
date_time	Unsigned 64-bit binary integer denoting a date and time as the number of elapsed 100-nanosecond units since 00:00 o'clock, November 17, 1858. This VMS data type is the same as the data type **absolute date and time** in Table 1–3.
device_name	Character string denoting the 1- to 15-character name of a device. It can be a logical name, but if it is, it must translate to a valid device name. If the device name contains a colon (:), the colon and the characters past it are ignored. When an underscore (_) precedes the device name string, it indicates that the string is a physical device name.
ef_cluster_name	Character string denoting the 1- to 15-character name of an event flag cluster. It can be a logical name, but if it is, it must translate to a valid event flag cluster name.
ef_number	Unsigned longword integer denoting the number of an event flag. Local event flags numbered 32 to 63 are available to your programs.
exit_handler_block	Variable-length structure denoting an exit handler control block. This control block, which describes the exit handler, is depicted in the following diagram:  ZK–1714–GE
fab	Structure denoting an RMS file access block.

31
0
Forward Link (Used by VMS only)
Exit Handler Address
These 3 bytes must be 0
arg. count
Address Condition Value (Written by VMS)
Additional argument for the exit handler; these are optional; one argument per longword

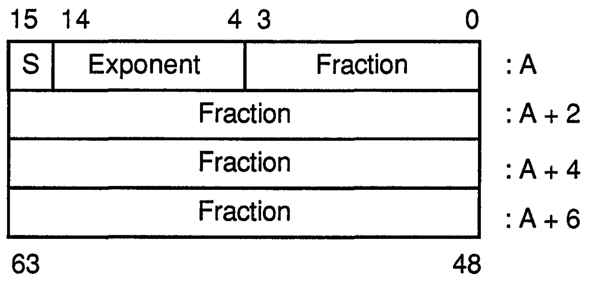
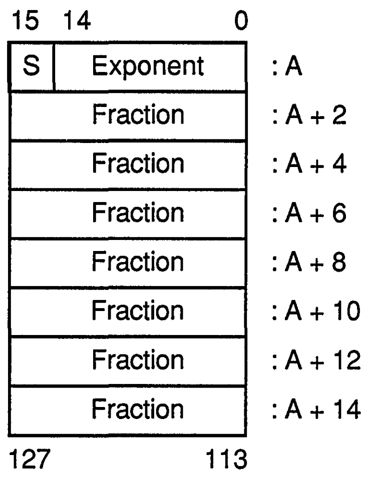
(continued on next page)

Table A–1 (Cont.) VMS Data Types

Data Type	Definition
file_protection	Unsigned word that is a 16-bit mask that specifies file protection. The mask contains four 4-bit fields, each of which specifies the protection to be applied to file access attempts by one of the four categories of users, from the rightmost field to the leftmost field: (1) system users, (2) the file owner, (3) users in the same UIC group as the owner, and (4) all other users (the world). Each field specifies, from the rightmost bit to the leftmost bit: (1) read access, (2) write access, (3) execute access, (4) delete access. Set bits indicate that access is denied. The following diagram depicts the 16-bit file-protection mask:  World Group Owner System D E W R D E W R D E W R D E W R 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 ZK–1706–GE
floating_point	One of the VAX standard floating-point data types. These types are F_floating, D_floating, G_floating, and H_floating. The structure of an F_floating number is as follows:  ZK–4197–GE The structure of a D_floating number is as follows:  15 14 7 6 0 S Exponent Fraction :A Fraction :A + 2 Fraction :A + 4 Fraction :A + 6 63 48 ZK–4198–GE

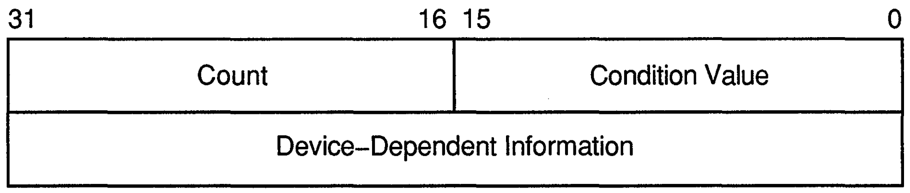
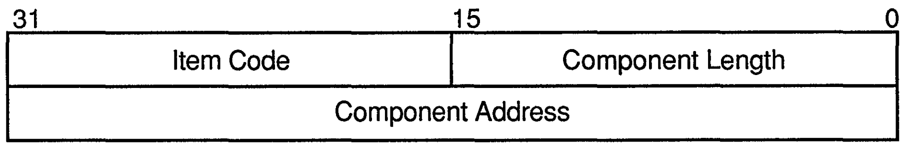
(continued on next page)

Table A–1 (Cont.) VMS Data Types

Data Type	Definition
	The structure of a G_floating number is as follows:  15 14 4 3 0 S Exponent Fraction :A Fraction :A + 2 Fraction :A + 4 Fraction :A + 6 63 48 ZK–4199–GE
	The structure of an H_floating number is as follows:  15 14 0 S Exponent :A Fraction :A + 2 Fraction :A + 4 Fraction :A + 6 Fraction :A + 8 Fraction :A + 10 Fraction :A + 12 Fraction :A + 14 127 113 ZK–4196–GE
function_code	Unsigned longword specifying the exact operations a procedure is to perform. This longword has two word-length fields: the first field is a number specifying the major operation; the second field is a mask or bit vector specifying various suboperations within the major operation.
identifier	Unsigned longword that identifies an object returned by the system.
io_status_block	Quadword structure containing information returned by a procedure that completes asynchronously. The information returned varies depending on the procedure.

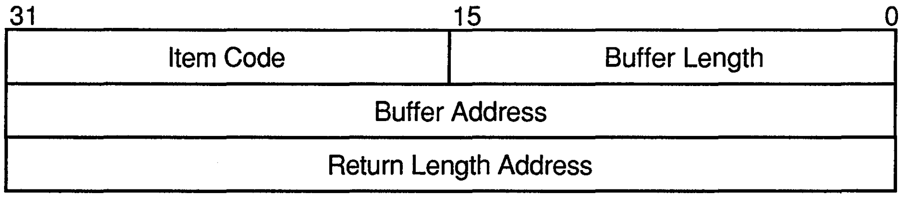
(continued on next page)

Table A–1 (Cont.) VMS Data Types

Data Type	Definition
	The following figure illustrates the format of the information written in the IOSB for SYS$QIO:  31 16 15 0 Count Condition Value Device–Dependent Information ZK–0856–GE The first word contains a condition value indicating the success or failure of the operation. The condition values used are the same as for all returns from system services; for example, SS$_NORMAL indicates successful completion. The second word contains the number of bytes actually transferred in the I/O operation. Note that for some devices this word contains only the low-order word of the count. The second longword contains device-dependent return information. To ensure successful I/O completion and the integrity of data transfers, you should check the IOSB following I/O requests, particularly for device-dependent I/O functions.
item_list_2	Structure that consists of one or more item descriptors and is terminated by a longword containing *0*. Each item descriptor is a 2-longword structure that contains three fields. The following diagram depicts a single item descriptor:  31 15 0 Item Code Component Length Component Address ZK–1709–GE The first field is a word in which the service writes the length (in characters) of the requested component. If the service does not locate the component, it returns the value *0* in this field and in the component address field. The second field contains a user-supplied, word-length symbolic code that specifies the component desired. The item codes are defined by the macros specific to the service. The third field is a longword in which the service writes the starting address of the component. This address is within the input string itself.
item_list_3	Structure that consists of one or more item descriptors and is terminated by a longword containing *0*. Each item descriptor is a 3-longword structure that contains four fields.

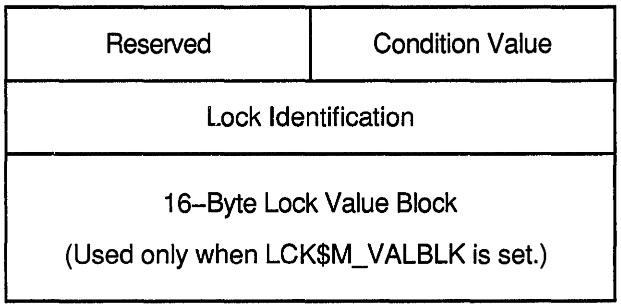
(continued on next page)

Table A–1 (Cont.) VMS Data Types

Data Type	Definition
	The following diagram depicts the format of a single item descriptor:  31 15 0 Item Code Buffer Length Buffer Address Return Length Address ZK–1705–GE The first field is a word containing a user-supplied integer specifying the length (in bytes) of the buffer in which the service writes the information. The length of the buffer needed depends upon the item code specified in the item code field of the item descriptor. If the value of buffer length is too small, the service truncates the data. The second field is a word containing a user-supplied symbolic code specifying the item of information that the service is to return. These codes are defined by macros specific to the service. The third field is a longword containing the user-supplied address of the buffer in which the service writes the information. The fourth field is a longword containing the user-supplied address of a word in which the service writes the length in bytes of the information it actually returned.
item_list_pair	Structure that consists of one or more longword pairs, or **doublets**, and is terminated by a longword containing *0*. Typically, the first longword contains an integer value such as a code. The second longword can contain a real or integer value.
item_quota_list	Structure that consists of one or more quota descriptors and is terminated by a byte containing a value defined by the symbolic name PQL$_LISTEND. Each quota descriptor consists of a 1-byte quota name followed by an unsigned longword containing the value for that quota.
lock_id	Unsigned longword integer denoting a lock identifier. This lock identifier is assigned to a lock by the lock manager facility when the lock is granted.
lock_status_block	Structure into which the lock manager facility writes status information about a lock. A lock status block always contains at least two longwords: the first word of the first longword contains a condition value; the second word of the first longword is reserved by Digital. The second longword also contains the lock identifier. The lock status block receives the final condition value plus the lock identification, and optionally contains a lock value block. When a request is queued, the lock identification is stored in the lock status block even if the lock has not been granted. This allows a procedure to dequeue locks that have not been granted. The condition value is placed in the lock status block only when the lock is granted (or when errors occur in granting the lock).

(continued on next page)

Table A–1 (Cont.) VMS Data Types

Data Type	Definition
	The following diagram depicts a lock status block that includes the optional 16-byte lock value block:  ZK–0376–GE
lock_value_block	16-byte block that the lock manager facility includes in a lock status block if the user requests it. The contents of the lock value block are user-defined and are not interpreted by the lock manager facility.
logical_name	Character string from 1 to 255 characters that identifies a logical name or equivalence name to be manipulated by VMS logical name system services. Logical names that denote specific VMS objects have their own VMS types; for example, a logical name identifying a device has the VMS type **device_name**.
longword_signed	Is the same as the data type **longword integer (signed)** in Table 1–3.
longword_unsigned	Is the same as the data type **longword (unsigned)** in Table 1–3.
mask_byte	Unsigned byte wherein each bit is interpreted by the called procedure. A mask is also referred to as a set of flags or as a bit mask.
mask_longword	Unsigned longword wherein each bit is interpreted by the called procedure. A mask is also referred to as a set of flags or as a bit mask.
mask_quadword	Unsigned quadword wherein each bit is interpreted by the called procedure. A mask is also referred to as a set of flags or as a bit mask.
mask_word	Unsigned word wherein each bit is interpreted by the called procedure. A mask is also referred to as a set of flags or as a bit mask.
null_arg	Unsigned longword denoting a null argument. A *null argument* is one whose only purpose is to hold a place in the argument list.
octaword_signed	Is the same as the data type **octaword integer (signed)** in Table 1–3.
octaword_unsigned	Is the same as the data type **octaword (unsigned)** in Table 1–3.

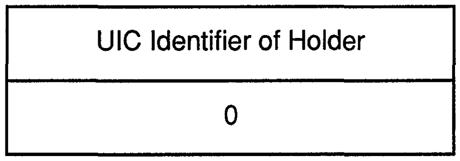
(continued on next page)

Table A–1 (Cont.) VMS Data Types

Data Type	Definition
page_protection	Unsigned longword specifying page protection to be applied by the VAX hardware. Protection values are specified using bits <3:0>; bits <31:4> are ignored. If you specify the protection as *0*, the protection defaults to kernel read only. The $PRTDEF macro defines the following symbolic names for the protection codes:

Symbol	Description
PRT$C_NA	No access
PRT$C_KR	Kernel read only
PRT$C_KW	Kernel write
PRT$C_ER	Executive read only
PRT$C_EW	Executive write
PRT$C_SR	Supervisor read only
PRT$C_SW	Supervisor write
PRT$C_UR	User read only
PRT$C_UW	User write
PRT$C_ERKW	Executive read; kernel write
PRT$C_SRKW	Supervisor read; kernel write
PRT$C_SREW	Supervisor read; executive write
PRT$C_URKW	User read; kernel write
PRT$C_UREW	User read; executive write
PRT$C_URSW	User read; supervisor write

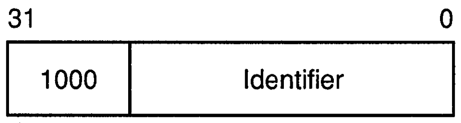
Data Type	Definition
procedure	Unsigned longword denoting the entry mask to a procedure that is not to be called at AST level. (Arguments specifying procedures to be called at AST level have the VMS type **ast_procedure**.)
process_id	Unsigned longword integer denoting a process identification (PID). This process identification is assigned to a process by VMS when the process is created.
process_name	Character string, containing 1 to 15 characters, that specifies the name of a process.
quadword_signed	Is the same as the data type **quadword integer (signed)** in 1–3.
quadword_unsigned	Is the same as the data type **quadword (unsigned)** in 1–3.
rights_holder	Unsigned quadword specifying a user's access rights to a system object. This quadword consists of two fields: the first is an unsigned longword identifier (VMS type **rights_id**), and the second is a longword bit mask wherein each bit specifies an access right. The following diagram depicts the format of a rights holder:



ZK–1903–GE

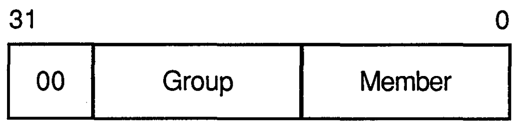
(continued on next page)

Table A–1 (Cont.) VMS Data Types

Data Type	Definition
rights_id	Unsigned longword denoting a rights identifier, which identifies an interest group in the context of the VMS security environment. This rights environment may consist of all or part of a user's user identification code (UIC). Identifiers have two formats in the rights database: UIC format (VMS type **uic**) and ID format. The high-order bits of the identifier value specify the format of the identifier. Two high-order zero bits identify a UIC format identifier; bit <31>, set to *1*, identifies an ID format identifier. Bit <31>, set to *1*, specifies ID format. Bits <30:28> are reserved by Digital. The remaining bits specify the identifier value. The following diagram depicts the ID format of a rights identifier:  ID Format ZK–1906–GE To the system, an identifier is a binary value; however, to make identifiers easy to use, the system translates the binary identifier value into an identifier name. The binary value and the identifier name are associated in the rights database. An identifier name consists of 1 to 31 alphanumeric characters and contains at least one nonnumeric character. An identifier name cannot consist entirely of numeric characters. It can include the characters A through Z, dollar signs ($), and underscores (_), as well as the numbers 0 through 9. Any lowercase characters are automatically converted to uppercase.
rab	Structure denoting an RMS record access block.
section_id	Unsigned quadword denoting a global section identifier. This identifier specifies the version of a global section and the criteria to be used in matching that global section.
section_name	Character string denoting a 1- to 43-character global-section name. This character string can be a logical name, but it must translate to a valid global-section name.
system_access_id	Unsigned quadword that denotes a system identification value to be associated with a rights database.
transaction_id	Unsigned octaword that denotes a unique transaction identifier.
time_name	Character string specifying a time value in VMS format.
uic	Unsigned longword denoting a user identification code (UIC). Each UIC is unique and represents a system user. The UIC identifier contains two high-order bits that designate format, a member field, and a group field. Member numbers range from 0 to 65,534; group numbers range from 1 to 16,382. The following diagram depicts the UIC format:

(continued on next page)

Table A–1 (Cont.) VMS Data Types

Data Type	Definition
	 UIC Format ZK–1905–GE
user_arg	Unsigned longword denoting a user-defined argument. This longword is passed to a procedure as an argument, but the contents of the longword are defined and interpreted by the user.
varying_arg	Unsigned longword denoting a variable argument. A variable argument can have variable types, depending on specifications made for other arguments in the call.
vector_byte_signed	A homogeneous array whose elements are all signed bytes.
vector_byte_unsigned	A homogeneous array whose elements are all unsigned bytes.
vector_longword_signed	A homogeneous array whose elements are all signed longwords.
vector_longword_unsigned	A homogeneous array whose elements are all unsigned longwords.
vector_quadword_signed	A homogeneous array whose elements are all signed quadwords.
vector_quadword_unsigned	A homogeneous array whose elements are all unsigned quadwords.
vector_word_signed	A homogeneous array whose elements are all signed words.
vector_word_unsigned	A homogeneous array whose elements are all unsigned words.
word_signed	Is the same as the data type **word integer (signed)** in Table 1–3.
word_unsigned	Is the same as the data type **word (unsigned)** in Table 1–3.

A.2 VAX Ada Implementation

Table A–2 lists the VMS data types and their corresponding VAX Ada data-type declarations.

Table A–2 VAX Ada Implementation

VMS Data Structure	VAX Ada Declaration
access_bit_names	STARLET.ACCESS_BIT_NAMES_TYPE
access_mode	STARLET.ACCESS_MODE_TYPE
address	SYSTEM.ADDRESS
address_range	STARLET.ADDRESS_RANGE_TYPE
arg_list	STARLET.ARG_LIST_TYPE
ast_procedure	SYSTEM.AST_HANDLER

(continued on next page)

Table A–2 (Cont.) VAX Ada Implementation

VMS Data Structure	VAX Ada Declaration
boolean	STANDARD.BOOLEAN
byte_signed	STANDARD.SHORT_SHORT_INTEGER
byte_unsigned	SYSTEM.UNSIGNED_BYTE
channel	STARLET.CHANNEL_TYPE
char_string	STANDARD.STRING
complex_number	User-defined record
cond_value	CONDITION_HANDLING.COND_VALUE_TYPE
context	STARLET.CONTEXT_TYPE
date_time	STARLET.DATE_TIME_TYPE
device_name	STARLET.DEVICE_NAME_TYPE
ef_cluster_name	STARLET.EF_CLUSTER_NAME_TYPE
ef_number	STARLET.EF_NUMBER_TYPE
exit_handler_block	STARLET.EXIT_HANDLER_BLOCK_TYPE
fab	STARLET.FAB_TYPE
file_protection	STARLET.FILE_PROTECTION_TYPE
floating_point	STANDARD.FLOAT STANDARD.LONG_FLOAT STANDARD.LONG_LONG_FLOAT SYSTEM.F_FLOAT SYSTEM.D_FLOAT SYSTEM.G_FLOAT SYSTEM.H_FLOAT
function_code	STARLET.FUNCTION_CODE_TYPE
identifier	SYSTEM.UNSIGNED_LONGWORD
io_status_block	STARLET.IOSB_TYPE
item_list_pair	SYSTEM.UNSIGNED_LONGWORD
item_list_2	STARLET.ITEM_LIST_2_TYPE
item_list_3	STARLET.ITEM_LIST_TYPE
item_quota_list	User-defined record
lock_id	STARLET.LOCK_ID_TYPE
lock_status_block	STARLET.LOCK_STATUS_BLOCK_TYPE
lock_value_block	STARLET.LOCK_VALUE_BLOCK_TYPE
logical_name	STARLET.LOGICAL_NAME_TYPE
longword_signed	STANDARD.INTEGER
longword_unsigned	SYSTEM.UNSIGNED_LONGWORD
mask_byte	SYSTEM.UNSIGNED_BYTE
mask_longword	SYSTEM.UNSIGNED_LONGWORD
mask_quadword	SYSTEM.UNSIGNED_QUADWORD

(continued on next page)

Table A–2 (Cont.) VAX Ada Implementation

VMS Data Structure	VAX Ada Declaration
mask_word	SYSTEM.UNSIGNED_WORD
null_arg	SYSTEM.UNSIGNED_LONGWORD
octaword_signed	**array**(1..4) **of** SYSTEM.UNSIGNED_LONGWORD
octaword_unsigned	**array**(1..4) **of** SYSTEM.UNSIGNED_LONGWORD
page_protection	STARLET.PAGE_PROTECTION_TYPE
procedure	SYSTEM.ADDRESS
process_id	STARLET.PROCESS_ID_TYPE
process_name	STARLET.PROCESS_NAME_TYPE
quadword_signed	SYSTEM.UNSIGNED_QUADWORD
quadword_unsigned	SYSTEM.UNSIGNED_QUADWORD
rights_holder	STARLET.RIGHTS_HOLDER_TYPE
rights_id	STARLET.RIGHTS_ID_TYPE
rab	STARLET.RAB_TYPE
section_id	STARLET.SECTION_ID_TYPE
section_name	STARLET.SECTION_NAME_TYPE
system_access_id	STARLET.SYSTEM_ACCESS_ID_TYPE
time_name	STARLET.TIME_NAME_TYPE
transaction_id	**array**(1..4) **of** SYSTEM.UNSIGNED_LONGWORD
uic	STARLET.UIC_TYPE
user_arg	STARLET.USER_ARG_TYPE
varying_arg	SYSTEM.UNSIGNED_LONGWORD
vector_byte_signed	**array**(1..n) **of** STANDARD.SHORT_SHORT_INTEGER
vector_byte_unsigned	**array**(1..n) **of** SYSTEM.UNSIGNED_BYTE
vector_longword_signed	**array**(1..n) **of** STANDARD.INTEGER
vector_longword_unsigned	**array**(1..n) **of** SYSTEM.UNSIGNED_LONGWORD
vector_quadword_signed	**array**(1..n) **of** SYSTEM.UNSIGNED_QUADWORD
vector_quadword_unsigned	**array**(1..n) **of** SYSTEM.UNSIGNED_QUADWORD
vector_word_signed	**array**(1..n) **of** STANDARD.SHORT_INTEGER
vector_word_unsigned	**array**(1..n) **of** SYSTEM.UNSIGNED_WORD
word_signed	STANDARD.SHORT_INTEGER
word_unsigned	SYSTEM.UNSIGNED_WORD

A.3 VAX APL Implementation

Table A–3 lists the VMS data types and their corresponding VAX APL data-type declarations.

Table A–3 VAX APL Implementation

VMS Data Type	VAX APL Declaration
access_bit_names	NA
access_mode	/TYPE=BU
address	NA
address_range	NA
arg_list	NA
ast_procedure	NA
boolean	/TYPE=V
byte_signed	/TYPE=B
byte_unsigned	/TYPE=BU
channel	/TYPE=WU
char_string	/TYPE=T
complex_number	/TYPE=FC /TYPE=DC /TYPE=GC /TYPE=HC
cond_value	/TYPE=LU
context	NA
date_time	NA
device_name	/TYPE=T
ef_cluster_name	/TYPE=T
ef_number	/TYPE=LU
exit_handler_block	NA
fab	NA
file_protection	/TYPE=WU
floating_point	/TYPE=F /TYPE=D /TYPE=G /TYPE=H
function_code	NA
identifier	NA
io_status_block	NA
item_list_2	NA
item_list_3	NA
item_list_pair	NA
item_quota_list	NA
lock_id	/TYPE=LU
lock_status_block	NA
lock_value_block	NA

(continued on next page)

Table A–3 (Cont.) VAX APL Implementation

VMS Data Type	VAX APL Declaration
logical_name	/TYPE=T
longword_signed	/TYPE=L
longword_unsigned	/TYPE=LU
mask_byte	/TYPE=BU
mask_longword	/TYPE=LU
mask_quadword	NA
mask_word	/TYPE=WU
null_arg	/TYPE=LU
octaword_signed	NA
octaword_unsigned	NA
page_protection	/TYPE=LU
procedure	NA
process_id	/TYPE=LU
process_name	/TYPE=T
quadword_signed	NA
quadword_unsigned	NA
rights_holder	NA
rights_id	/TYPE=LU
rab	NA
section_id	NA
section_name	/TYPE=T
system_access_id	NA
time_name	/TYPE=T
transaction_id	NA
uic	/TYPE=LU
user_arg	/TYPE=LU
varying_arg	NA
vector_byte_signed	/TYPE=B
vector_byte_unsigned	/TYPE=BU
vector_longword_signed	/TYPE=L
vector_longword_unsigned	/TYPE=LU
vector_quadword_signed	NA
vector_quadword_unsigned	NA
vector_word_signed	/TYPE=W

(continued on next page)

Table A–3 (Cont.) VAX APL Implementation

VMS Data Type	VAX APL Declaration
vector_word_unsigned	/TYPE=WU
word_signed	/TYPE=W
word_unsigned	/TYPE=WU

A.4 VAX BASIC Implementation

Table A–4 lists the VMS data types and their corresponding VAX BASIC data-type declarations.

Table A–4 VAX BASIC Implementation

VMS Data Type	VAX BASIC Declaration
access_bit_names	NA
access_mode	BYTE (signed)
address	LONG

address_range	LONG address_range (1)
	or
	RECORD address_range
	    LONG beginning_address
	    LONG ending_address
	END RECORD

arg_list	NA
ast_procedure	EXTERNAL LONG ast_proc
boolean	LONG
byte_signed	BYTE
byte_unsigned	BYTE[1]
channel	WORD
char_string	STRING

complex_number	RECORD complex
	    REAL real_part
	    REAL imaginary_part
	END RECORD

cond_value	LONG
context	LONG

date_time	RECORD date_time
	    LONG FILL (2)
	END RECORD

device_name	STRING

[1]Although unsigned data types are not directly supported in VAX BASIC, you may substitute the signed equivalent provided you do not exceed the range of the signed data type.

(continued on next page)

Table A–4 (Cont.) VAX BASIC Implementation

VMS Data Type	VAX BASIC Declaration
ef_cluster_name	STRING
ef_number	LONG

exit_handler_block	RECORD EHCB
	    LONG flink
	    LONG handler_addr
	    BYTE arg_count
	    BYTE FILL (3)
	    LONG status_value_addr
	END RECORD

fab	NA
file_protection	LONG
floating_point	SINGLE DOUBLE GFLOAT HFLOAT

function_code	RECORD function-code
	    WORD major-function
	    WORD subfunction
	END RECORD

identifier	LONG

io_status_block	RECORD iosb
	    WORD iosb-field (3)
	END RECORD

item_list_2	RECORD item_list_two
	    GROUP item(15)
	      VARIANT
	      CASE
	        WORD comp_length
	        WORD code
	        LONG comp_address
	      CASE
	        LONG terminator
	      END VARIANT
	    END GROUP
	END RECORD

(continued on next page)

Table A–4 (Cont.) VAX BASIC Implementation

VMS Data Type	VAX BASIC Declaration
item_list_3	RECORD item_list_3 / GROUP item (15) / VARIANT / CASE / WORD buf_len / WORD code / LONG buffer_address / LONG length_address / CASE / LONG terminator / END VARIANT / END GROUP / END RECORD
item_list_pair	RECORD item_list_pair / GROUP item (15) / VARIANT / CASE / LONG code / LONG value / CASE / LONG terminator / END VARIANT / END GROUP / END RECORD item_list_pair
item_quota_list	RECORD item_quota_list / GROUP quota (n) / VARIANT / CASE / BYTE quota_name / LONG value / CASE / BYTE list_end / END VARIANT / END GROUP / END RECORD
lock_id	LONG
lock_status_block	NA
lock_value_block	NA
logical_name	STRING
longword_signed	LONG
longword_unsigned	LONG[1]
mask_byte	BYTE
mask_longword	LONG

item_list_3:

```
RECORD item_list_3
          GROUP item ( 15 )
             VARIANT
             CASE
                WORD buf_len
                WORD code
                LONG buffer_address
                LONG length_address
             CASE
                LONG terminator
             END VARIANT
          END GROUP
END RECORD
```

item_list_pair:

```
RECORD item_list_pair
          GROUP item (15)
             VARIANT
             CASE
                LONG code
                LONG value
             CASE
                LONG terminator
             END VARIANT
          END GROUP
END RECORD item_list_pair
```

item_quota_list:

```
RECORD item_quota_list
          GROUP quota ( n )
             VARIANT
             CASE
                BYTE quota_name
                LONG value
             CASE
                BYTE list_end
             END VARIANT
          END GROUP
END RECORD
```

[1]Although unsigned data types are not directly supported in VAX BASIC, you may substitute the signed equivalent provided you do not exceed the range of the signed data type.

(continued on next page)

Table A–4 (Cont.) VAX BASIC Implementation

VMS Data Type	VAX BASIC Declaration
mask_quadword	RECORD quadword LONG FILL (2) END RECORD[1]
mask_word	WORD
null_arg	A null argument is indicated by a comma used as a placeholder in the argument list.
octaword_signed	NA
octaword_unsigned	NA
page_protection	LONG
procedure	EXTERNAL LONG proc
process_id	LONG
process_name	STRING
quadword_signed	RECORD quadword LONG FILL (2) END RECORD
quadword_unsigned	RECORD quadword LONG FILL (2) END RECORD[1]
rights_holder	RECORD quadword LONG FILL (2) END RECORD[1]
rights_id	LONG
rab	NA
section_id	RECORD quadword LONG FILL (2) END RECORD[1]
section_name	STRING
system_access_id	RECORD quadword LONG FILL (2) END RECORD[1]
time_name	STRING
transaction_id	NA
uic	LONG
user_arg	LONG
varying_arg	Dependent upon application.
vector_byte_signed	BYTE array (n)
vector_byte_unsigned	BYTE array (n)[1]
vector_longword_signed	LONG array (n)

[1]Although unsigned data types are not directly supported in VAX BASIC, you may substitute the signed equivalent provided you do not exceed the range of the signed data type.

(continued on next page)

Table A–4 (Cont.) VAX BASIC Implementation

VMS Data Type	VAX BASIC Declaration
vector_longword_unsigned	LONG array (n)[1]
vector_quadword_signed	NA
vector_quadword_unsigned	NA
vector_word_signed	WORD array (n)
vector_word_unsigned	WORD array (n)[1]
word_signed	WORD
word_unsigned	WORD[1]

[1]Although unsigned data types are not directly supported in VAX BASIC, you may substitute the signed equivalent provided you do not exceed the range of the signed data type.

A.5 VAX BLISS Implementation

Table A–5 lists the VMS data types and their corresponding VAX BLISS data-type declarations.

Table A–5 VAX BLISS Implementation

VMS Data Type	VAX BLISS Declaration
access_bit_names	BLOCKVECTOR[32,8,BYTE]
access_mode	UNSIGNED BYTE
address	UNSIGNED LONG
address_range	VECTOR[2,LONG,UNSIGNED]
arg_list	VECTOR[n,LONG,UNSIGNED] where *n* is the number of arguments + 1.
ast_procedure	UNSIGNED LONG
boolean	UNSIGNED LONG
byte_signed	SIGNED BYTE
byte_unsigned	UNSIGNED BYTE
channel	UNSIGNED WORD
char_string	VECTOR[65536,BYTE,UNSIGNED]
complex_number	F_Complex: VECTOR[2,LONG] D_Complex: VECTOR[4,LONG] G_Complex: VECTOR[4,LONG] H_Complex: VECTOR[8,LONG]
cond_value	UNSIGNED LONG
context	UNSIGNED LONG
date_time	VECTOR[2,LONG,UNSIGNED]

(continued on next page)

Table A–5 (Cont.) VAX BLISS Implementation

VMS Data Type	VAX BLISS Declaration
device_name	VECTOR[n,BYTE,UNSIGNED] where n is the length of the device name.
ef_cluster_name	VECTOR[n,BYTE,UNSIGNED] where n is the length of the event flag cluster name.
ef_number	UNSIGNED LONG
exit_handler_block	BLOCK[n,BYTE] where n is the size of the exit handler control block.
fab	$FAB_DECL (from STARLET.REQ)
file_protection	BLOCK[2,BYTE]
floating_point	F_Floating: VECTOR[1,LONG] D_Floating: VECTOR[2,LONG] G_Floating: VECTOR[2,LONG] H_Floating: VECTOR[4,LONG]
function_code	BLOCK[2,WORD]
identifier	UNSIGNED LONG
io_status_block	BLOCK[8,BYTE]
item_list_2	BLOCKVECTOR[n,8,BYTE] where n is the number of the item descriptors + 1.
item_list_3	BLOCKVECTOR[n,12,BYTE] where n is the number of the item descriptors + 1. $ITMLST_DECL/$ITMLST_INIT from STARLET.REQ
item_list_pair	BLOCKVECTOR[n,2,LONG] where n is the number of the item descriptors + 1.
item_quota_list	BLOCKVECTOR[n,5,BYTE] where n is the number of the quota descriptors + 1.
lock_id	UNSIGNED_LONG
lock_status_block	BLOCK[n,BYTE] where n is the size of the lock_status_block minus at least 8.
lock_value_block	BLOCK[16,BYTE]
logical_name	VECTOR[255,BYTE,UNSIGNED]
longword_signed	SIGNED LONG
longword_unsigned	UNSIGNED LONG
mask_byte	BITVECTOR[8]

(continued on next page)

Table A–5 (Cont.) VAX BLISS Implementation

VMS Data Type	VAX BLISS Declaration
mask_longword	BITVECTOR[32]
mask_quadword	BITVECTOR[64]
mask_word	BITVECTOR[16]
null_arg	UNSIGNED LONG
octaword_signed	VECTOR[4,LONG,UNSIGNED]
octaword_unsigned	VECTOR[4,LONG,UNSIGNED]
page_protection	UNSIGNED LONG
procedure	UNSIGNED LONG
process_id	UNSIGNED LONG
process_name	VECTOR[n,BYTE,UNSIGNED] where n is the length of the process name.
quadword_signed	VECTOR[2,LONG,UNSIGNED]
quadword_unsigned	VECTOR[2,LONG,UNSIGNED]
rights_holder	BLOCK[8,BYTE]
rights_id	UNSIGNED LONG
rab	$RAB_DECL from STARLET.REQ
section_id	VECTOR[2,LONG,UNSIGNED]
section_name	VECTOR[n,BYTE,UNSIGNED] where n is the length of the global section name.
system_access_id	VECTOR[2,LONG,UNSIGNED]
time_name	VECTOR[n,BYTE,UNSIGNED] where n is the length of the time value in VMS format.
transaction_id	VECTOR[4,LONG,UNSIGNED]
uic	UNSIGNED LONG
user_arg	UNSIGNED LONG
varying_arg	UNSIGNED LONG
vector_byte_signed	VECTOR[n,BYTE,SIGNED] where n is the size of the array.
vector_byte_unsigned	VECTOR[n,BYTE,UNSIGNED] where n is the size of the array.
vector_longword_signed	VECTOR[n,LONG,SIGNED] where n is the size of the array.

(continued on next page)

Table A–5 (Cont.) VAX BLISS Implementation

VMS Data Type	VAX BLISS Declaration
vector_longword_unsigned	VECTOR[n,LONG,UNSIGNED] where n is the size of the array.
vector_quadword_signed	BLOCKVECTOR[n,2,LONG] where n is the size of the array.
vector_quadword_unsigned	BLOCKVECTOR[n,2,LONG] where n is the size of the array.
vector_word_signed	VECTOR[n,BYTE,SIGNED] where n is the size of the array.
vector_word_unsigned	VECTOR[n,BYTE,UNSIGNED] where n is the size of the array.
word_signed	SIGNED WORD
word_unsigned	UNSIGNED WORD

A.6 VAX C Implementation

Table A–6 lists the VMS data types and their corresponding VAX C data-type declarations.

Table A–6 VAX C Implementation

VMS Data Type	VAX C Declaration
access_bit_names	User-defined[1]
access_mode	unsigned char
address	int *pointer[2,4]
address_range	int *array [2][2,3,4]
arg_list	User-defined[1]
ast_procedure	Pointer to function[2]
boolean	unsigned long int
byte_signed	char
byte_unsigned	unsigned char
channel	unsigned short int
char_string	char array[n][3,5]
complex_number	User-defined[1]

[1]The declaration of a user-defined data structure depends on how the data will be used. Such data structures can be declared in a variety of ways, each of which is suitable only to specific applications.

[2]The term **pointer** refers to several declarations involving pointers. Pointers are declared with special syntax and associated with the data type of the object being pointed to. This object is often *user-defined.*

[3]The term **array** denotes the syntax of a VAX C array declaration.

[4]The data type specified can be changed to any valid VAX C data type.

[5]The size of the array must be substituted for *n.*

(continued on next page)

Table A–6 (Cont.) VAX C Implementation

VMS Data Type	VAX C Declaration
cond_value	unsigned long int
context	unsigned long int
date_time	User-defined[1]
device_name	char array[n][3,5]
ef_cluster_name	char array[n][3,5]
ef_number	unsigned long int
exit_handler_block	User-defined[1]
fab	#include fab from text library struct FAB
file_protection	unsigned short int or user-defined[1]
floating_point	float or double
function_code	unsigned long int or user-defined[1]
identifier	int *pointer[2,4]
io_status_block	User-defined[1]
item_list_2	User-defined[1]
item_list_3	User-defined[1]
item_list_pair	User-defined[1]
item_quota_list	User-defined[1]
lock_id	unsigned long int
lock_status_block	User-defined[1]
lock_value_block	User-defined[1]
logical_name	char array[n][3,5]
longword_signed	long int
longword_unsigned	unsigned long int
mask_byte	unsigned char
mask_longword	unsigned long int
mask_quadword	User-defined[1]
mask_word	unsigned short int
null_arg	unsigned long int
octaword_signed	User-defined[1]
octaword_unsigned	User-defined[1]

[1]The declaration of a user-defined data structure depends on how the data will be used. Such data structures can be declared in a variety of ways, each of which is suitable only to specific applications.

[2]The term **pointer** refers to several declarations involving pointers. Pointers are declared with special syntax and associated with the data type of the object being pointed to. This object is often *user-defined.*

[3]The term **array** denotes the syntax of a VAX C array declaration.

[4]The data type specified can be changed to any valid VAX C data type.

[5]The size of the array must be substituted for *n*.

(continued on next page)

Table A–6 (Cont.) VAX C Implementation

VMS Data Type	VAX C Declaration
page_protection	unsigned long int
procedure	Pointer to function[2]
process_id	unsigned long int
process_name	char array[n][3,5]
quadword_signed	User-defined[1]
quadword_unsigned	User-defined[1]
rights_holder	User-defined[1]
rights_id	unsigned long int
rab	#include rab from text library struct RAB
section_id	User-defined[1]
section_name	char array[n][3,5]
system_access_id	User-defined[1]
time_name	char array[n][3,5]
transaction_id	User-defined[1]
uic	unsigned long int
user_arg	User-defined[1]
varying_arg	User-defined[1]
vector_byte_signed	char array[n][3,5]
vector_byte_unsigned	unsigned char array[n][3,5]
vector_longword_signed	long int array[n][3,5]
vector_longword_unsigned	unsigned long int array[n][3,5]
vector_quadword_signed	User-defined[1]
vector_quadword_unsigned	User-defined[1]
vector_word_signed	short int array[n][3,5]
vector_word_unsigned	unsigned short int array[n][3,5]
word_signed	short int
word_unsigned	unsigned short int

[1]The declaration of a user-defined data structure depends on how the data will be used. Such data structures can be declared in a variety of ways, each of which is suitable only to specific applications.

[2]The term **pointer** refers to several declarations involving pointers. Pointers are declared with special syntax and associated with the data type of the object being pointed to. This object is often *user-defined.*

[3]The term **array** denotes the syntax of a VAX C array declaration.

[5]The size of the array must be substituted for *n*.

A.7 VAX COBOL Implementation

Table A–7 lists the VMS data types and their corresponding VAX COBOL data-type declarations.

Table A–7 VAX COBOL Implementation

VMS Data Type	VAX COBOL Declaration
access_bit_names	NA . . . PIC X(128)[2]
access_mode	NA . . . PIC X[2] access_mode is usually passed BY VALUE as PIC 9(5) COMP
address	USAGE POINTER
address_range	01 ADDRESS-RANGE 02 BEGINNING-ADDRESS USAGE POINTER 02 ENDING-ADDRESS USAGE POINTER
arg_list	NA . . . Constructed by the compiler as a result of using the COBOL CALL statement. An argument list may be created as follows, but cannot be referenced by the COBOL CALL statement. 01 ARG-LIST 02 ARG-COUNT PIC S9(5) COMP 02 ARG-BY-VALUE PIC S9(5) COMP 02 ARG-BY-REFERENCE USAGE POINTER 02 VALUE REFERENCE ARG-NAME . . . continue as needed
ast_procedure	01 AST-PROC PIC 9(5) COMP[1]
boolean	01 BOOLEAN-VALUE PIC 9(5) COMP[1]
byte_signed	NA . . . PIC X[2]
byte_unsigned	NA . . . PIC X[2]
channel	01 CHANNEL PIC 9(4) COMP[1]
char_string	01 CHAR-STRING PIC X to PIC X(65535)
complex_number	NA . . . PIC X(n) where *n* is length[2]
cond_value	01 COND-VALUE PIC 9(5) COMP[1]
context	01 CONTEXT PIC 9(5) COMP[1]
date_time	NA . . . PIC X(16)[2]
device_name	01 DEVICE-NAME PIC X(n) where *n* is length.
ef_cluster_name	01 CLUSTER-NAME PIC X(n) where *n* is length.
ef_number	01 EF-NO PIC 9(5) COMP[1]

[1]Although unsigned computational data structures are not directly supported in VAX COBOL, you may substitute the signed equivalent provided you do not exceed the range of the signed data structure.

[2]Most VMS data types not directly supported in VAX COBOL can be represented as an alphanumeric data item of a certain number of bytes. While VAX COBOL does not interpret the data type, it may be used to pass objects from one language to another.

(continued on next page)

Table A–7 (Cont.) VAX COBOL Implementation

VMS Data Type	VAX COBOL Declaration
exit_handler_block	NA . . . PIC X(n) where *n* is length[2]
fab	NA . . . Too complex for general COBOL use. Most of a FAB structure can be described by a lengthy COBOL record description, but such a FAB cannot then be referenced by a COBOL I-O statement. It is much simpler to do the I-O completely within COBOL, and let the COBOL compiler generate the FAB structure, or do the I-O in another language.
file_protection	01 FILE-PROT PIC 9(4) COMP[1]
floating_point	01 F-FLOAT USAGE COMP-1
	01 D-FLOAT USAGE COMP-2
	g-float and h-float are not supported in VAX COBOL.
function_code	01 FUNCTION-CODE
	02 MAJOR-FUNCTION PIC 9(4) COMP[1]
	02 SUB-FUNCTION PIC 9(4) COMP[1]
identifier	01 ID PIC 9(5) COMP[1]
io_status_block	01 IOSB
	02 COND-VAL PIC 9(4) COMP[1]
	02 BYTE-CNT PIC 9(4) COMP[1]
	02 DEV-INFO PIC 9(5) COMP[1]
item_list_2	01 ITEM-LIST-TWO
	02 ITEM-LIST OCCURS n TIMES
	04 COMP-LENGTH PIC S9(4) COMP
	04 ITEM-CODE PIC S9(4) COMP
	04 COMP-ADDRESS PIC S9(5) COMP
	02 TERMINATOR PIC S9(5) COMP VALUE 0
item_list_3	01 ITEM-LIST-3
	02 ITEM-LIST OCCURS n TIMES
	04 BUF-LEN PIC S9(4) COMP
	04 ITEM-CODE PIC S9(4) COMP
	04 BUFFER-ADDRESS PIC S9(5) COMP
	04 LENGTH-ADDRESS PIC S9(5) COMP
	02 TERMINATOR PIC S9(5) COMP VALUE 0
item_list_pair	01 ITEM-LIST-PAIR
	02 ITEM-LIST OCCURS n TIMES
	04 ITEM-CODE PIC S9(5) COMP
	04 ITEM-VALUE PIC S9(5) COMP
	02 TERMINATOR PIC S9(5) COMP VALUE 0
item_quota_list	NA

[1]Although unsigned computational data structures are not directly supported in VAX COBOL, you may substitute the signed equivalent provided you do not exceed the range of the signed data structure.

[2]Most VMS data types not directly supported in VAX COBOL can be represented as an alphanumeric data item of a certain number of bytes. While VAX COBOL does not interpret the data type, it may be used to pass objects from one language to another.

(continued on next page)

Table A–7 (Cont.) VAX COBOL Implementation

VMS Data Type	VAX COBOL Declaration
lock_id	01 LOCK-ID PIC 9(5) COMP[1]
lock_status_block	NA
lock_value_block	NA
logical_name	01 LOG-NAME PIC X TO X(255)
longword_signed	01 LWS PIC S9(5) COMP
longword_unsigned	01 LWU PIC 9(5) COMP[1]
mask_byte	NA . . . PIC X[2]
mask_longword	01 MLW PIC 9(5) COMP[1]
mask_quadword	01 MQW PIC 9(18) COMP[1]
mask_word	01 MW PIC 9(4) COMP[1]
null_arg	CALL . . . USING OMITTED or PIC S9(5) COMP VALUE 0 passed USING BY VALUE
octaword_signed	NA
octaword_unsigned	NA
page_protection	01 PAGE-PROT PIC 9(5) COMP[1]
procedure	01 ENTRY-MASK PIC 9(5) COMP[1]
process_id	01 PID PIC 9(5) COMP[1]
process_name	01 PROCESS-NAME PIC X TO X(15)
quadword_signed	01 QWS PIC S9(18) COMP
quadword_unsigned	01 QWU PIC 9(18) COMP[1]
rights_holder	01 RIGHTS-HOLDER 02 RIGHTS-ID PIC 9(5) COMP[1] 02 ACCESS-RIGHTS PIC 9(5) COMP[1]
rights_id	01 RIGHTS-ID PIC 9(5) COMP[1]
rab	NA . . . Too complex for general COBOL use. Most of a RAB structure can be described by a lengthy COBOL record description, but such a RAB cannot then be referenced by a COBOL I-O statement. It is much simpler to do the I-O completely within COBOL, and let the COBOL compiler generate the RAB structure, or do the I-O in another language.
section_id	01 SECTION-ID PIC 9(18) COMP[1]
section_name	01 SECTION-NAME PIC X to X(43)
system_access_id	01 SECTION-ACCESS-ID PIC 9(18) COMP[1]

[1]Although unsigned computational data structures are not directly supported in VAX COBOL, you may substitute the signed equivalent provided you do not exceed the range of the signed data structure.

[2]Most VMS data types not directly supported in VAX COBOL can be represented as an alphanumeric data item of a certain number of bytes. While VAX COBOL does not interpret the data type, it may be used to pass objects from one language to another.

(continued on next page)

Table A–7 (Cont.) VAX COBOL Implementation

VMS Data Type	VAX COBOL Declaration
time_name	01 TIME-NAME PIC X(n) where *n* is the length.
transaction_id	NA
uic	01 UIC PIC 9(5) COMP[1]
user_arg	01 USER-ARG PIC 9(5) COMP[1]
varying_arg	Dependent upon application
vector_byte_signed	NA . . . [3]
vector_byte_unsigned	NA . . . [3]
vector_longword_signed	NA . . . [3]
vector_longword_unsigned	NA . . . [3]
vector_quadword_signed	NA . . . [3]
vector_quadword_unsigned	NA . . . [3]
vector_word_signed	NA . . . [3]
vector_word_unsigned	NA . . . [3]
word_signed	01 WS PIC S9(4) COMP
word_unsigned	01 WS PIC 9(4) COMP[1]

[1]Although unsigned computational data structures are not directly supported in VAX COBOL, you may substitute the signed equivalent provided you do not exceed the range of the signed data structure.

[3]VAX COBOL does not permit the passing of variable-length data structures.

A.8 VAX FORTRAN Implementation

Table A–8 lists the VMS data types and their corresponding VAX FORTRAN data-type declarations.

Table A–8 VAX FORTRAN Implementation

VMS Data Type	VAX FORTRAN Declaration
access_bit_names	INTEGER*4(2,32) or STRUCTURE /access_bit_names/ INTEGER*4 access_name_len INTEGER*4 access_name_buf END STRUCTURE !access_bit_names RECORD /access_bit_names/ my_names(32)
access_mode	BYTE
address	INTEGER*4

(continued on next page)

Table A–8 (Cont.) VAX FORTRAN Implementation

VMS Data Type	VAX FORTRAN Declaration

address_range	INTEGER*4(2)
	or
	STRUCTURE /address_range/
	        INTEGER*4 low_address
	        INTEGER*4 high_address
	END STRUCTURE

arg_list	INTEGER*4(n)
ast_procedure	EXTERNAL
boolean	LOGICAL*4
byte_signed	BYTE
byte_unsigned	BYTE[1]
channel	INTEGER*2
char_string	CHARACTER*n
complex_number	COMPLEX*8 COMPLEX*16
cond_value	INTEGER*4
context	INTEGER*4
date_time	INTEGER*4(2)
device_name	CHARACTER*n
ef_cluster_name	CHARACTER*n
ef_number	INTEGER*4

exit_handler_block	STRUCTURE /exhblock/
	        INTEGER*4 flink
	        INTEGER*4 exit_handler_addr
	        BYTE(3) /0/
	        BYTE arg_count
	        INTEGER*4 cond_value
	        ! .
	        ! .(optional arguments . . .
	        ! . one argument per longword)
	        !
	END STRUCTURE !cntrlblk
	
	RECORD /exhblock/ myexh_block

fab	INCLUDE '($FABDEF)' RECORD /fabdef/ myfab
file_protection	INTEGER*4

[1]Unsigned data types are not directly supported by VAX FORTRAN. However, in most cases you can substitute the signed equivalent as long as you do not exceed the range of the signed data structure.

(continued on next page)

Table A–8 (Cont.) VAX FORTRAN Implementation

VMS Data Type	VAX FORTRAN Declaration
floating_point	REAL*4 REAL*8 DOUBLE PRECISION REAL*16
function_code	INTEGER*4
identifier	INTEGER*4

io_status_block	STRUCTURE /iosb/
	    INTEGER*2 iostat, !return status
	    2 term_offset, !loc. of line terminator
	    2 terminator, !value of terminator
	    2 term_size !size of terminator
	END STRUCTURE
	
	RECORD /iosb/ my_iosb

item_list_2	STRUCTURE /itmlst/
	    UNION
	    MAP
	    INTEGER*2 buflen,code
	    INTEGER*4 bufadr
	    END MAP
	    MAP
	    INTEGER*4 end_list /0/
	    END MAP
	    END UNION
	END STRUCTURE !itmlst
	
	RECORD /itmlst/ my_itmlst_2(n)
	(Allocate *n* records where *n* is the number of item codes plus an extra element for the end-of-list item.)

item_list_3	STRUCTURE /itmlst/
	    UNION
	    MAP
	    INTEGER*2 buflen,code
	    INTEGER*4 bufadr,retlenadr
	    END MAP
	    MAP
	    INTEGER*4 end_list /0/
	    END MAP
	    END UNION
	END STRUCTURE !itmlst
	
	RECORD /itmlst/ my_itmlst_2(n)
	(Allocate *n* records where *n* is the number of item codes plus an extra element for the end-of-list item.)

(continued on next page)

Table A–8 (Cont.) VAX FORTRAN Implementation

VMS Data Type	VAX FORTRAN Declaration

item_list_pair	STRUCTURE /itmlist_pair/
	    UNION
	    MAP
	      INTEGER*4 code
	      INTEGER*4 value
	    END MAP
	    MAP
	      INTEGER*4 end_list /0/
	    END MAP
	    END UNION
	END STRUCTURE !itmlst_pair
	
	RECORD /itmlst_pair/ my_itmlst_pair(n)
	(Allocate *n* records where *n* is the number of item codes plus an extra element for the end-of-list item.)

item_quota_list	STRUCTURE /item_quota_list/
	    MAP
	    BYTE quota_name
	    INTEGER*4 quota_value
	    END MAP
	    MAP
	    BYTE end_quota_list
	    END MAP
	END STRUCTURE !item_quota_list

lock_id	INTEGER*4

lock_status_block	STRUCTURE/lksb/
	    INTEGER*2 cond_value
	    INTEGER*2 unused
	    INTEGER*4 lock_id
	    BYTE(16)
	END STRUCTURE !lock_status_lock

lock_value_block	BYTE(16)
logical_name	CHARACTER*n
longword_signed	INTEGER*4
longword_unsigned	INTEGER*4[1]
mask_byte	INTEGER*1
mask_longword	INTEGER*4
mask_quadword	INTEGER*4(2)
mask_word	INTEGER*2
null_arg	%VAL(0)
octaword_signed	INTEGER*4(4)

[1]Unsigned data types are not directly supported by VAX FORTRAN. However, in most cases you can substitute the signed equivalent as long as you do not exceed the range of the signed data structure.

(continued on next page)

Table A–8 (Cont.) VAX FORTRAN Implementation

VMS Data Type	VAX FORTRAN Declaration
octaword_unsigned	INTEGER*4(4)[1]
page_protection	INTEGER*4
procedure	INTEGER*4
process_id	INTEGER*4
process_name	CHARACTER*n
quadword_signed	INTEGER*4(2)
quadword_unsigned	INTEGER*4(2)[1]
rights_holder	INTEGER*4(2) or STRUCTURE /rights_holder/ INTEGER*4 rights_id INTEGER*4 rights_mask END STRUCTURE !rights_holder
rights_id	INTEGER*4
rab	INCLUDE '($RABDEF)' RECORD /rabdef/ myrab
section_id	INTEGER*4(2)
section_name	CHARACTER*n
system_access_id	INTEGER*4(2)
time_name	CHARACTER*23
transaction_id	INTEGER*4(4)[1]
uic	INTEGER*4
user_arg	Any longword quantity
varying_arg	INTEGER*4
vector_byte_signed	BYTE(n)
vector_byte_unsigned	BYTE(n)[1]
vector_longword_signed	INTEGER*4(n)
vector_longword_unsigned	INTEGER*4(n)[1]
vector_quadword_signed	INTEGER*4(2, n)
vector_quadword_unsigned	INTEGER*4(2,n)[1]
vector_word_signed	INTEGER*2(n)
vector_word_unsigned	INTEGER*2(n)[1]
word_signed	INTEGER*2(n)
word_unsigned	INTEGER*2(n)[1]

[1]Unsigned data types are not directly supported by VAX FORTRAN. However, in most cases you can substitute the signed equivalent as long as you do not exceed the range of the signed data structure.

A.9 VAX MACRO Implementation

Table A–9 lists the VMS data types and their corresponding VAX MACRO data-type declarations.

Table A–9 VAX MACRO Implementation

VMS Data Type	VAX MACRO Declaration
access_bit_names	.ASCID /name_for_bit0/ .ASCID /name_for_bit1/ . . . .ASCID /name_for_bit31/
access_mode	.BYTE PSL$C_xxxx
address	.ADDRESSS virtual_address
address_range	.ADDRESS start_address,end_address
arg_list	.LONG n_args, arg1, arg2, . . .
ast_procedure	.ADDRESS ast_procedure
boolean	.LONG 1 or .LONG 0
byte_signed	.SIGNED_BYTE byte_value
byte_unsigned	.BYTE byte_value
channel	.WORD channel_number
char_string	.ASCID /string/
complex_number	NA
cond_value	.LONG cond_value
context	.LONG 0
date_time	.QUAD date_time
device_name	.ASCID /ddcu:/
ef_cluster_name	.ASCID /ef_cluster_name/
ef_number	.LONG ef_number
exit_handler_block	.LONG 0 .ADDRESS exit_handler_routine .LONG 1 .ADDRESS status STATUS: .BLKL 1
fab	MYFAB: $FAB
file_protection	.WORD prot_value
floating_point	.FLOAT, .G_FLOAT, or .H_FLOAT
function_code	.LONG code!mask
identifier	.ADDRESSS virtual_address
io_status_block	.QUAD 0
item_list_2	.WORD component_length .WORD item_code .ADDRESS component_address

(continued on next page)

Table A–9 (Cont.) VAX MACRO Implementation

VMS Data Type	VAX MACRO Declaration
item_list_3	.WORD buffer_length .WORD item_code .ADDRESS buffer_address .ADDRESS return_length_address
item_list_pair	.LONG item_code .LONG data
item_quota_list	.BYTE PQL$_xxxx .LONG value_for_quota .BYTE pql$_listend
lock_id	.LONG lock_id
lock_status_block	.QUAD 0
lock_value_block	.BLKB 16
logical_name	.ASCID /logical_name/
longword_signed	.LONG value
longword_unsigned	.LONG value
mask_byte	.BYTE mask_byte
mask_longword	.LONG mask_longword
mask_quadword	.QUAD mask_quadword
mask_word	.WORD mask_word
null_arg	.LONG 0
octaword_signed	NA
octaword_unsigned	.OCTA value
page_protection	.LONG page_protection
procedure	.ADDRESS procedure
process_id	.LONG process_id
process_name	.ASCID /process_name/
quadword_signed	NA
quadword_unsigned	.QUAD value
rights_holder	.LONG identifier, access_rights_bitmask
rights_id	.LONG rights_id
rab	MYRAB: $RAB
section_id	.LONG sec$k_matXXX, version_number
section_name	.ASCID /section_name/
system_access_id	.QUAD system_access_id
time_name	.ASCID /dd-mmm-yyyy:hh:mm:ss.cc/
transaction_id	.OCTA value
uic	.LONG uic
user_arg	.LONG data

(continued on next page)

Table A–9 (Cont.) VAX MACRO Implementation

VMS Data Type	VAX MACRO Declaration
varying_arg	Dependent upon application
vector_byte_signed	.SIGNED_BYTE val1,val2, . . . valN
vector_byte_unsigned	.BYTE val1,val2, . . . valN
vector_longword_signed	.LONG val1,val2, . . . valN
vector_longword_unsigned	.LONG val1,val2, . . . valN
vector_quadword_signed	NA
vector_quadword_unsigned	.QUAD val1, val2, . . . valN
vector_word_signed	.SIGNED_WORD val1,val2, . . . valN
vector_word_unsigned	.WORD val1,val2, . . . valN
word_signed	.SIGNED_WORD value
word_unsigned	.WORD value

A.10 VAX Pascal Implementation

Table A–10 lists the VMS data types and their corresponding VAX Pascal data-type declarations.

Table A–10 VAX Pascal Implementation

VMS Data Type	VAX Pascal Declaration
access_bit_names	PACKED ARRAY [1..32] OF [QUAD] RECORD END;[1,6]
access_mode	[BYTE] 0..3;[6]
address	UNSIGNED;
address_range	PACKED ARRAY [1..2] OF UNSIGNED;[6]
arg_list	PACKED ARRAY [1..n] OF UNSIGNED;[6]
ast_procedure	UNSIGNED;
boolean	BOOLEAN;[3]
byte_signed	[BYTE] -128..127;[6]
byte_unsigned	[BYTE] 0..255;[6]
channel	[WORD] 0..65535;[6]
char_string	[CLASS_S] PACKED ARRAY [L..U:INTEGER] OF CHAR;[4]

[1]This type is not available in VAX Pascal when an empty record has been inserted. To manipulate the contents, declare with explicit field components. If you pass an empty record as a parameter to a Pascal routine, you must use the VAR keyword.

[3]VAX Pascal allocates a byte for a BOOLEAN variable. Use the [LONG] attribute when passing to routines that expect a longword.

[4]This parameter declaration accepts VARYING OF CHAR or PACKED ARRAY OF CHAR and produces the CLASS_S descriptor required by system services.

[6]VAX Pascal expects either a type identifier or conformant schema. Declare this under the TYPE declaration and use the type identifier in the formal parameter declaration.

(continued on next page)

Table A–10 (Cont.) VAX Pascal Implementation

VMS Data Type	VAX Pascal Declaration
complex_number	[LONG(2)] RECORD END; * F_Floating Complex *[1,6] [QUAD(2)] RECORD END; * D/G_Floating Complex * [OCTA(2)] RECORD END; * H_Floating Complex *
cond_value	UNSIGNED;
context	UNSIGNED;
date_time	[QUAD] RECORD END;[1,6]
device_name	[CLASS_S] PACKED ARRAY [L..U:INTEGER] OF CHAR;[4]
ef_cluster_name	[CLASS_S] PACKED ARRAY [L..U:INTEGER] OF CHAR;[4]
ef_number	UNSIGNED;
exit_handler_block	PACKED ARRAY [1..n] OF UNSIGNED;[6]
fab	FAB$TYPE;[5]
file_protection	[WORD] RECORD END;[1,6]

floating_point	REAL; { F_Floating }
	SINGLE; { F_Floating }
	DOUBLE; { D_Floating/G_Floating }[2]
	QUADRUPLE; { H_Floating }

function_code	UNSIGNED;
identifier	UNSIGNED;
io_status_block	[QUAD] RECORD END;[1,6]

item_list_2	PACKED ARRAY [1..n] OF PACKED RECORD[6]
	    CASE INTEGER OF
	    1: (
	    FIELD1 : [WORD] 0..65535;
	    FIELD2 : [WORD] 0..65535;
	    FIELD3 : UNSIGNED);
	    2: (
	    TERMINATOR : UNSIGNED);
	    END;

[1]This type is not available in VAX Pascal when an empty record has been inserted. To manipulate the contents, declare with explicit field components. If you pass an empty record as a parameter to a Pascal routine, you must use the VAR keyword.

[2]If the [G_FLOATING] attribute is used in compiling, double-precision variables and expressions are represented in G_floating format. The /G_FLOATING command line qualifier can also be used. Both methods default to no G_floating.

[4]This parameter declaration accepts VARYING OF CHAR or PACKED ARRAY OF CHAR and produces the CLASS_S descriptor required by system services.

[5]The program must inherit the STARLET environment file located in SYS$LIBRARY:STARLET.PEN.

[6]VAX Pascal expects either a type identifier or conformant schema. Declare this under the TYPE declaration and use the type identifier in the formal parameter declaration.

(continued on next page)

Table A–10 (Cont.) VAX Pascal Implementation

VMS Data Type	VAX Pascal Declaration

item_list_3	PACKED ARRAY [1..n] OF PACKED RECORD[6]
	CASE INTEGER OF
	1: (
	FIELD1 : [WORD] 0..65535;
	FIELD2 : [WORD] 0..65535;
	FIELD3 : UNSIGNED;
	FIELD4 : UNSIGNED);
	2: (
	TERMINATOR : UNSIGNED);
	END;

item_list_pair	PACKED ARRAY [1..n] OF PACKED RECORD[6]
	CASE INTEGER OF
	1: (
	FIELD1 : INTEGER;
	FIELD2 : INTEGER);
	2: (
	TERMINATOR : UNSIGNED);
	END;

item_quota_list	PACKED ARRAY [1..n] OF PACKED RECORD[6]
	CASE INTEGER OF
	1: (
	QUOTA_NAME : [BYTE] 0..255;
	QUOTA_VALUE: UNSIGNED);
	2: (
	QUOTA_TERM : [BYTE] 0..255);
	END;

lock_id	UNSIGNED;
lock_status_block	[BYTE(24)] RECORD END;[1,6]
lock_value_block	[BYTE(16)] RECORD END;[1,6]
logical_name	[CLASS_S] PACKED ARRAY [L..U:INTEGER] OF CHAR;[4]
longword_signed	INTEGER;
longword_unsigned	UNSIGNED;
mask_byte	[BYTE,UNSAFE] PACKED ARRAY [1..8] OF BOOLEAN;[6]
mask_longword	[LONG,UNSAFE] PACKED ARRAY [1..32] OF BOOLEAN;[6]
mask_quadword	[QUAD,UNSAFE] PACKED ARRAY [1..64] OF BOOLEAN;[6]
mask_word	[WORD,UNSAFE] PACKED ARRAY [1..16] OF BOOLEAN;[6]
null_arg	UNSIGNED;
octaword_signed	[OCTA] RECORD END;[1,6]

[1]This type is not available in VAX Pascal when an empty record has been inserted. To manipulate the contents, declare with explicit field components. If you pass an empty record as a parameter to a Pascal routine, you must use the VAR keyword.

[4]This parameter declaration accepts VARYING OF CHAR or PACKED ARRAY OF CHAR and produces the CLASS_S descriptor required by system services.

[6]VAX Pascal expects either a type identifier or conformant schema. Declare this under the TYPE declaration and use the type identifier in the formal parameter declaration.

(continued on next page)

Table A–10 (Cont.) VAX Pascal Implementation

VMS Data Type	VAX Pascal Declaration
octaword_unsigned	[OCTA] RECORD END;[1,6]
page_protection	[LONG] 0..7;[6]
procedure	UNSIGNED;
process_id	UNSIGNED;
process_name	[CLASS_S] PACKED ARRAY [L..U:INTEGER] OF CHAR;[4]
quadword_signed	[QUAD] RECORD END;[1,6]
quadword_unsigned	[QUAD] RECORD END;[1,6]
rights_holder	[QUAD] RECORD END;[1,6]
rights_id	UNSIGNED;
rab	RAB$TYPE;[5]
section_id	[QUAD] RECORD END;[1,6]
section_name	[CLASS_S] PACKED ARRAY [L..U:INTEGER] OF CHAR;[4]
system_access_id	[QUAD] RECORD END;[1,6]
time_name	[CLASS_S] PACKED ARRAY [L..U:INTEGER] OF CHAR;[4]
transaction_id	[OCTA] RECORD END;[1,6]
uic	UNSIGNED;
user_arg	[UNSAFE] UNSIGNED;
varying_arg	[UNSAFE,REFERENCE] PACKED ARRAY [L..U:INTEGER] OF [BYTE] 0..255;
vector_byte_signed	PACKED ARRAY [1..n] OF [BYTE] -128..127;[6]
vector_byte_unsigned	PACKED ARRAY [1..n] OF [BYTE] 0..255;[6]
vector_longword_signed	PACKED ARRAY [1..n] OF INTEGER;[6]
vector_longword_unsigned	PACKED ARRAY [1..n] OF UNSIGNED;[6]
vector_quadword_signed	PACKED ARRAY [1..n] OF [QUAD] RECORD END;[1,6]
vector_quadword_unsigned	PACKED ARRAY [1..n] OF [QUAD] RECORD END;[1,6]
vector_word_signed	PACKED ARRAY [1..n] OF [WORD] -32768..32767;[6]
vector_word_unsigned	PACKED ARRAY [1..n] OF [WORD] 0..65535;[6]
word_signed	[WORD] -32768..32767;[6]
word_unsigned	[WORD] 0..65535;[6]

[1]This type is not available in VAX Pascal when an empty record has been inserted. To manipulate the contents, declare with explicit field components. If you pass an empty record as a parameter to a Pascal routine, you must use the VAR keyword.

[4]This parameter declaration accepts VARYING OF CHAR or PACKED ARRAY OF CHAR and produces the CLASS_S descriptor required by system services.

[5]The program must inherit the STARLET environment file located in SYS$LIBRARY:STARLET.PEN.

[6]VAX Pascal expects either a type identifier or conformant schema. Declare this under the TYPE declaration and use the type identifier in the formal parameter declaration.

A.11 VAX PL/I Implementation

Table A–11 lists the VMS data types and their corresponding VAX PL/I data-type declarations.

Table A–11 VAX PL/I Implementation

VMS Data Type	VAX PL/I Declaration

access_bit_names	1 ACCESS_BIT_NAMES(32),
	    2 LENGTH FIXED BINARY(15),
	    2 DTYPE FIXED BINARY(7)
	    INITIAL((32)DSC$K_DTYPE_T),
	    2 CLASS FIXED BINARY(7)
	    INITIAL((32)DSC$K_CLASS_S),
	    2 CHAR_PTR POINTER;[1]
	
	The length of the LENGTH field in each element of the array should correspond to the length of a string of characters pointed to by the CHAR_PTR field. The constants DSC$K_CLASS_S and DSC$K_DTYPE_T can be used by including the module $DSCDEF from PLI$STARLET.

access_mode	FIXED BINARY(7) (The constants for this type: PSLC_KERNEL, PSLC_EXEC, PSLC_SUPER, PSLC_USER—are declared in module $PSLDEF in PLI$STARLET.)
address	POINTER
address_range	(2) POINTER[1]

arg_list	1 ARG_LIST BASED,
	    2 ARGCOUNT FIXED BINARY(31),
	    2 ARGUMENT (X REFER (ARGCOUNT))
	    POINTER;[1]
	
	If the arguments are passed by value, you may need to change the type of the ARGUMENT field of the structure. Alternatively, you can use the POSINT, INT, or UNSPEC built-in functions /pseudovariables to access the data. X should be an expression with a value in the range 0 to 255 when the structure is allocated.

ast_procedure	PROCEDURE or ENTRY[2]
boolean	BIT ALIGNED[1]

[1]System routines are often written so the parameter passed occupies more storage than the object requires. For example, some system services have parameters that return a bit value as a longword. These variables must be declared BIT(32) ALIGNED (not BIT(n) ALIGNED) so adjacent storage is not overwritten by return values or used incorrectly as input. (Longword parameters are always declared BIT(32) ALIGNED.)

[2]AST procedures and those passed as parameters of type ENTRY VALUE or ANY VALUE must be external procedures. This applies to all system routines that take procedure parameters.

(continued on next page)

Table A–11 (Cont.) VAX PL/I Implementation

VMS Data Type	VAX PL/I Declaration
byte_signed	FIXED BINARY(7)
byte_unsigned	FIXED BINARY(7)[3]
channel	FIXED BINARY(15)
char_string	CHARACTER(n)[5]
complex_number	(2) FLOAT BINARY(n) (See floating_point for values of *n*.)
cond_value	See STS$VALUE in module $STSDEF in PLI$STARLET.[1]
context	FIXED BINARY(31)
date_time	BIT(64) ALIGNED[4,6]
device_name	CHARACTER(n)[5]
ef_cluster_name	CHARACTER(n)[5]
ef_number	FIXED BINARY(31)

exit_handler_block	1 EXIT_HANDLER_BLOCK BASED,
	    2 FORWARD_LINK POINTER,
	    2 HANDLER POINTER,
	    2 ARGCOUNT FIXED BINARY(31),
	    2 ARGUMENT (n REFER
	    (ARGCOUNT)) POINTER;[1]
	(Replace *n* with an expression that yields a value between 0 and 255 when the structure is allocated.)

fab	See module $FABDEF in PLI$STARLET.
file_protection	BIT(16) ALIGNED[1]

[1]System routines are often written so the parameter passed occupies more storage than the object requires. For example, some system services have parameters that return a bit value as a longword. These variables must be declared BIT(32) ALIGNED (not BIT(n) ALIGNED) so adjacent storage is not overwritten by return values or used incorrectly as input. (Longword parameters are always declared BIT(32) ALIGNED.)

[3]This is actually an unsigned integer. This declaration is interpreted as a signed number; use the POSINT function to determine the actual value.

[4]VAX PL/I does not support FIXED BINARY numbers with precisions greater than 31. To use larger values, declare variables to be BIT variables of the appropriate size and use the POSINT and SUBSTR bits as necessary to access the values, or declare the item as a structure. The RTL routines LIB$ADDX and LIB$SUBX may be useful if you need to perform arithmetic on these types.

[5]System services require CHARACTER string representation for parameters. Most other system routines allow either CHARACTER or CHARACTER VARYING. For parameter declarations, *n* should be an asterisk (*).

[6]Routines declared in PLI$STARLET often use ANY so you are free to declare the data structure in the most convenient way for the application. ANY may be necessary in some cases because PL/I does not allow parameter declarations for some data types used by VMS. (In particular, PL/I parameters with arrays passed by reference cannot be declared to have nonconstant bounds.)

(continued on next page)

Table A–11 (Cont.) VAX PL/I Implementation

VMS Data Type	VAX PL/I Declaration
floating_point	FLOAT BINARY(n) The values for *n* are as follows: 1 <= n <= 24 — F floating 25 <= n <= 53 — D floating 25 <= n <= 53 — G floating (with /G_FLOAT) 54 <= n <= 113 — H floating
function_code	BIT(32) ALIGNED
identifier	POINTER

io_status_block	Because there are different formats for I/O status blocks for various system services, different definitions are appropriate for different uses. Some of the common formats are as follows:
	
	/* See the *VMS System Services Reference Manual.* */
	1 IOSB_SYS$GETSYI,
	        2 STATUS FIXED BINARY(31),
	        2 RESERVED FIXED BINARY(31);
	
	/* See the *VMS I/O User's Reference Manual: Part I.* */
	1 IOSB_TTDRIVER_A,
	        2 STATUS FIXED BINARY(15),
	        2 BYTE_COUNT FIXED BINARY(15),
	        2 MBZ FIXED BINARY(31) INITIAL(0);
	
	/* See the *VMS I/O User's Reference Manual: Part I.* */
	1 IOSB_TTDRIVER_B,
	        2 STATUS FIXED BINARY(15),
	        2 TRANSMIT_SPEED FIXED BINARY(7),
	        2 RECEIVE_SPEED FIXED BINARY(7),
	        2 CR_FILL FIXED BINARY(7),
	        2 LF_FILL FIXED BINARY(7),
	        2 PARITY_FLAGS FIXED BINARY(7),
	        2 MBZ FIXED BINARY(7) INITIAL(0);

(continued on next page)

Table A–11 (Cont.) VAX PL/I Implementation

VMS Data Type	VAX PL/I Declaration

item_list_2	`1 ITEM_LIST_2,`
	`    2 ITEM(SIZE),`
	`      3 COMPONENT_LENGTH FIXED`
	`      BINARY(15),`
	`      3 ITEM_CODE FIXED BINARY(15),`
	`      3 COMPONENT_ADDRESS`
	`      POINTER,`
	`    2 TERMINATOR FIXED BINARY(31)`
	`    INITIAL(0);`[1]
	
	(Replace SIZE with the number of items you want.)

item_list_3	`1 ITEM_LIST_3,`
	`    2 ITEM(SIZE),`
	`      3 BUFFER_LENGTH FIXED`
	`      BINARY(15),`
	`      3 ITEM_CODE FIXED BINARY(15),`
	`      3 BUFFER_ADDRESS POINTER,`
	`      3 RETURN_LENGTH POINTER,`
	`    2 TERMINATOR FIXED BINARY(31)`
	`    INITIAL(0);`[1]
	
	(Replace SIZE with the number of items you want.)

item_list_pair	`1 ITEM_LIST_PAIR,`
	`    2 ITEM(SIZE),`
	`      3 ITEM_CODE FIXED BINARY(31),`
	`      3 ITEM UNION,`
	`        4 INTEGER FIXED BINARY(31),`
	`        4 REAL FLOAT BINARY(24),`
	`    2 TERMINATOR FIXED BINARY(31)`
	`    INITIAL(0);`[1]
	
	(Replace SIZE with the number of items you want.)

[1]System routines are often written so the parameter passed occupies more storage than the object requires. For example, some system services have parameters that return a bit value as a longword. These variables must be declared BIT(32) ALIGNED (not BIT(n) ALIGNED) so adjacent storage is not overwritten by return values or used incorrectly as input. (Longword parameters are always declared BIT(32) ALIGNED.)

(continued on next page)

Table A–11 (Cont.) VAX PL/I Implementation

VMS Data Type	VAX PL/I Declaration
item_quota_list	1 ITEM_QUOTA_LIST, 2 QUOTA(SIZE), 3 NAME FIXED BINARY(7), 3 VALUE FIXED BINARY(31), 2 TERMINATOR FIXED BINARY(7) INITIAL(PQL$_LISTEND);[1] (Replace SIZE with the number of quota entries you want to use. The constant PQL$_LISTEND can be used by including the module $PQLDEF from PLI$STARLET or by declaring it GLOBALREF FIXED BINARY(31) VALUE.)
lock_id	FIXED BINARY(31)
lock_status_block	1 LOCK_STATUS_BLOCK, 2 STATUS_CODE FIXED BINARY(15), 2 RESERVED FIXED BINARY(15), 2 LOCK_ID FIXED BINARY(31);[1]
lock_value_block	The declaration of an item of this structure depends on the use of the structure, because VMS does not interpret the value.[1]
logical_name	CHARACTER(n)[5]
longword_signed	FIXED BINARY(31)
longword_unsigned	FIXED BINARY(31)[3]
mask_byte	BIT(8) ALIGNED
mask_longword	BIT(32) ALIGNED
mask_quadword	BIT(64) ALIGNED
mask_word	BIT(16) ALIGNED
null_arg	Omit the corresponding parameter in the call. For example, FOO(A,,B) would omit the second parameter.

[1]System routines are often written so the parameter passed occupies more storage than the object requires. For example, some system services have parameters that return a bit value as a longword. These variables must be declared BIT(32) ALIGNED (not BIT(n) ALIGNED) so adjacent storage is not overwritten by return values or used incorrectly as input. (Longword parameters are always declared BIT(32) ALIGNED.)

[3]This is actually an unsigned integer. This declaration is interpreted as a signed number; use the POSINT function to determine the actual value.

[5]System services require CHARACTER string representation for parameters. Most other system routines allow either CHARACTER or CHARACTER VARYING. For parameter declarations, *n* should be an asterisk (*).

(continued on next page)

Table A–11 (Cont.) VAX PL/I Implementation

VMS Data Type	VAX PL/I Declaration
octaword_signed	BIT(128) ALIGNED[4,6]
octaword_unsigned	BIT(128) ALIGNED[4,6]
page_protection	FIXED BINARY(31) (The constants for this type are declared in module $PRTDEF in PLI$STARLET.)
procedure	PROCEDURE or ENTRY[2]
process_id	FIXED BINARY(31)
process_name	CHARACTER(n)[5]
quadword_signed	BIT(64) ALIGNED[4,6]
quadword_unsigned	BIT(64) ALIGNED[4,6]
rights_holder	1 RIGHTS_HOLDER, 2 RIGHTS_ID FIXED BINARY(31), 2 ACCESS_RIGHTS BIT(32) ALIGNED;[1]
rights_id	FIXED BINARY(31)
rab	See module $RABDEF in PLI$STARLET[1]
section_id	BIT(64) ALIGNED
section_name	CHARACTER(n)[5]
system_access_id	BIT(64) ALIGNED
time_name	CHARACTER(n)[5]
transaction_id	BIT(128) ALIGNED[4,6]
uic	FIXED BINARY(31)
user_arg	ANY
varying_arg	ANY with OPTIONS(VARIABLE) on the routine declaration or with OPTIONAL on the parameter declaration.

[1]System routines are often written so the parameter passed occupies more storage than the object requires. For example, some system services have parameters that return a bit value as a longword. These variables must be declared BIT(32) ALIGNED (not BIT(n) ALIGNED) so adjacent storage is not overwritten by return values or used incorrectly as input. (Longword parameters are always declared BIT(32) ALIGNED.)

[2]AST procedures and those passed as parameters of type ENTRY VALUE or ANY VALUE must be external procedures. This applies to all system routines that take procedure parameters.

[4]VAX PL/I does not support FIXED BINARY numbers with precisions greater than 31. To use larger values, declare variables to be BIT variables of the appropriate size and use the POSINT and SUBSTR bits as necessary to access the values, or declare the item as a structure. The RTL routines LIB$ADDX and LIB$SUBX may be useful if you need to perform arithmetic on these types.

[5]System services require CHARACTER string representation for parameters. Most other system routines allow either CHARACTER or CHARACTER VARYING. For parameter declarations, *n* should be an asterisk (*).

[6]Routines declared in PLI$STARLET often use ANY so you are free to declare the data structure in the most convenient way for the application. ANY may be necessary in some cases because PL/I does not allow parameter declarations for some data types used by VMS. (In particular, PL/I parameters with arrays passed by reference cannot be declared to have nonconstant bounds.)

(continued on next page)

Table A–11 (Cont.) VAX PL/I Implementation

VMS Data Type	VAX PL/I Declaration
vector_byte_signed	(n) FIXED BINARY(7)[7]
vector_byte_unsigned	(n) FIXED BINARY(7)[3,7]
vector_longword_signed	(n) FIXED BINARY(31)[7]
vector_longword_unsigned	(n) FIXED BINARY(31)[3,7]
vector_quadword_signed	(n) BIT(64) ALIGNED[4,6,7]
vector_quadword_unsigned	(n) BIT(64) ALIGNED[3,4,6,7]
vector_word_signed	(n) FIXED BINARY(15)[7]
vector_word_unsigned	(n) FIXED BINARY(15)[3,7]
word_signed	FIXED BINARY(15)
word_unsigned	FIXED BINARY(15)[4]

[3]This is actually an unsigned integer. This declaration is interpreted as a signed number; use the POSINT function to determine the actual value.

[4]VAX PL/I does not support FIXED BINARY numbers with precisions greater than 31. To use larger values, declare variables to be BIT variables of the appropriate size and use the POSINT and SUBSTR bits as necessary to access the values, or declare the item as a structure. The RTL routines LIB$ADDX and LIB$SUBX may be useful if you need to perform arithmetic on these types.

[6]Routines declared in PLI$STARLET often use ANY so you are free to declare the data structure in the most convenient way for the application. ANY may be necessary in some cases because PL/I does not allow parameter declarations for some data types used by VMS. (In particular, PL/I parameters with arrays passed by reference cannot be declared to have nonconstant bounds.)

[7]For parameter declarations, the bounds must be constant for arrays passed by reference. For arrays passed by descriptor, *s should be used for the array extent instead. (VMS system routines almost always take arrays by reference.)

Note: All system services and many system constants and data structures are declared in PLI$STARLET.TLB.

While the current version of VAX PL/I does not support unsigned fixed binary numbers or fixed binary numbers with a precision greater than 31, future versions may support these features. If VAX PL/I is extended to support these types, declarations in PLISTARLET may change to use the new data types where appropriate.

A.12 VAX RPG II Implementation

Table A–12 lists the VMS data types and their corresponding VAX RPG II data-type declarations.

Table A–12 VAX RPG II Implementation

VMS Data Type	VAX RPG II Declaration
access_bit_names	NA
access_mode	Declare as text string of one byte. When using this data structure, you must interpret the ASCII contents of the string to determine the **access_mode**.
address	L[1]
address_range	Q[1]
arg_list	NA
ast_procedure	L[1]
boolean	NA
byte_signed	Declare as text string of one byte. When using this data structure, you must interpret the ASCII contents of the string.
byte_unsigned	Same as for **byte_signed.**[1]
channel	W[1]
char_string	TEXT STRING
complex_number	DATA STRUCTURE
cond_value	cond_value GIVNG OPCODE
context	L[1]
date_time	Q[1]
device_name	TEXT STRING
ef_cluster_name	TEXT STRING
ef_number	L[1]
exit_handler_block	DATA STRUCTURE
fab	Implicitly generated by the compiler on your behalf. You cannot access the fab data structure from an RPG II program.
file_protection	W[1]
floating_point	F D
function_code	F
identifier	L[1]
io_status_block	Q
item_list_pair	DATA STRUCTURE
item_list_2	DATA STRUCTURE
item_list_3	DATA STRUCTURE

[1]Technically, RPG II does not support unsigned data structures. However, unsigned information may be passed using the signed equivalent, provided the contents do not exceed the range of the signed data structure.

(continued on next page)

Table A–12 (Cont.) VAX RPG II Implementation

VMS Data Type	VAX RPG II Declaration
item_quota_list	NA
lock_id	L[1]
lock_status_block	DATA STRUCTURE
lock_value_block	DATA STRUCTURE
logical_name	TEXT STRING
longword_signed	L
longword_unsigned	L[1]
mask_byte	Same as for byte_signed[1]
mask_longword	L[1]
mask_quadword	Q[1]
mask_word	W[1]
null_arg	NA
octaword_signed	DATA STRUCTURE
octaword_unsigned	DATA STRUCTURE
page_protection	L[1]
procedure	L[1]
process_id	L[1]
process_name	TEXT STRING
quadword_signed	Q
quadword_unsigned	Q[1]
rights_holder	Q[1]
rights_id	L[1]
rab	Implicitly generated by the compiler on your behalf. You cannot access the **rab** data structure from an RPG II program.
section_id	Q[1]
section_name	TEXT STRING
system_access_id	Q[1]
time_name	TEXT STRING
transaction_id	DATA STRUCTURE
uic	L[1]
user_arg	L[1]
varying_arg	Dependent upon application
vector_byte_signed	ARRAY OF TEXT STRING
vector_byte_unsigned	ARRAY OF TEXT STRING[1]

[1]Technically, RPG II does not support unsigned data structures. However, unsigned information may be passed using the signed equivalent, provided the contents do not exceed the range of the signed data structure.

(continued on next page)

Table A–12 (Cont.) VAX RPG II Implementation

VMS Data Type	VAX RPG II Declaration
vector_longword_signed	ARRAY OF LONGWORD INTEGER (SIGNED) L
vector_longword_unsigned	RAY OF LONGWORD INTEGER L[1]
vector_quadword_signed	NA
vector_quadword_unsigned	NA
vector_word_signed	ARRAY OF WORD INTEGER (SIGNED) W
vector_word_unsigned	ARRAY OF WORD INTEGER W[1]
word_signed	W
word_unsigned	W[1]

[1]Technically, RPG II does not support unsigned data structures. However, unsigned information may be passed using the signed equivalent, provided the contents do not exceed the range of the signed data structure.

A.13 VAX SCAN Implementation

Table A–13 lists the VMS data types and their corresponding VAX SCAN data-type declarations.

Table A–13 VAX SCAN Implementation

VMS Data Type	VAX SCAN Declaration
access_bit_name	FILL(8*32)[1]
access_mode	FILL(1)[1]
address	POINTER

address_range	RECORD
	start: POINTER,
	end: POINTER,
	END RECORD

arg_list	RECORD count: INTEGER, arg1: POINTER, ! if by reference arg2: INTEGER, ! if by value . . . ! depending on needs END RECORD
ast_procedure	POINTER
boolean	BOOLEAN[2]

[1]FILL is a data type that can always be used. A FILL is an object between 0 and 65K bytes in length. VAX SCAN does not interpret the contents of an object. Thus, it can be used to pass or return the object to another language that does understand the type.

[2]SCAN Boolean is just one byte.

(continued on next page)

Table A–13 (Cont.) VAX SCAN Implementation

VMS Data Type	VAX SCAN Declaration
byte_signed	FILL(1)[1]
byte_unsigned	FILL(1)[1]
channel	FILL(2)[1]
char_string	FIXED STRING(x) where *x* is length
complex_number	FILL(x) where *x* is length[1]
cond_value	INTEGER
context	INTEGER
date_time	FILL(8)[1]
device_name	FIXED STRING(x) where *x* is length
ef_cluster_name	FIXED STRING(x) where *x* is length
ef_number	INTEGER
exit_handler_block	FILL(x) where *x* is length[1]
fab	A **fab** is too complicated a structure to include in this chart—much of it can be described with a SCAN record; however, it is much simpler and less prone to error to access **fabs** from other languages that have the record predefined.
file_protection	FILL(2)[1]
floating_point	FILL(x) where *x* is length[1]
function_code	INTEGER
identifier	POINTER
io_status_block	FILL(8)[1]

item_list_2	RECORD
	    item1: FILL(8),
	    item2: FILL(8),
	    . . .
	    terminator: INTEGER,
	END RECORD[1]

item_list_3	RECORD
	    item1: FILL(12),
	    item2: FILL(12),
	    . . .
	    terminator: INTEGER,
	END RECORD[1]

[1]FILL is a data type that can always be used. A FILL is an object between 0 and 65K bytes in length. VAX SCAN does not interpret the contents of an object. Thus, it can be used to pass or return the object to another language that does understand the type.

(continued on next page)

Table A–13 (Cont.) VAX SCAN Implementation

VMS Data Type	VAX SCAN Declaration

item_list_pair	RECORD
	    pair_1: RECORD ! 2 integer pair
	        long1: INTEGER,
	        long2: INTEGER,
	        END RECORD,
	    pair_2: RECORD ! integer-real pair
	        long1: INTEGER,
	        long2: FILL(4),
	        END RECORD,
	    . . . ! depending on need
	    terminator: INTEGER,
	END RECORD

item_quota_list	RECORD
	    item1: RECORD
	        type: FILL(1),
	        value: INTEGER,
	    END RECORD,
	    item2: RECORD
	        type: FILL(1),
	        value: INTEGER,
	    END RECORD,
	    . . .
	    terminator: FILL(1),
	END RECORD[1]

lock_id	INTEGER

lock_status_block	RECORD
	    status: FILL(2),
	    reserved: FILL(2),
	    lock_id: INTEGER,
	END RECORD[1]

lock_value_block	FILL(16)[1]
logical_name	FIXED STRING(x) where *x* is length
longword_signed	INTEGER
longword_unsigned	INTEGER
mask_byte	FILL(1)[1]
mask_longword	INTEGER

mask_quadword	RECORD
	    first_half: INTEGER,
	    second_half: INTEGER,
	END RECORD

mask_word	FILL(2)[1]
null_arg	Use asterisk (*) for argument.

[1] FILL is a data type that can always be used. A FILL is an object between 0 and 65K bytes in length. VAX SCAN does not interpret the contents of an object. Thus, it can be used to pass or return the object to another language that does understand the type.

(continued on next page)

Table A–13 (Cont.) VAX SCAN Implementation

VMS Data Type	VAX SCAN Declaration
octaword_signed	FILL(16)[1]
octaword_unsigned	FILL(16)[1]
page_protection	INTEGER
procedure	POINTER
process_id	INTEGER
process_name	FIXED STRING(x) where *x* is length
quadword_signed	FILL(8)[1]
quadword_unsigned	FILL(8)[1]
rights_holder	RECORD rights_id: INTEGER, bitmask: INTEGER, END RECORD
rights_id	INTEGER
rab	A **rab** is too complicated a structure to include in this chart—much of it can be described with a SCAN record; however, it is much simpler and less prone to error to access **rabs** from other languages that have the record predefined.
second_name	FILL(8)[1]
section_name	FIXED STRING(x) where *x* is length
system_access_id	FILL(8)[1]
time_name	FIXED STRING(x) where *x* is length
transaction_id	FILL(16)[1]
uic	INTEGER
user_arg	INTEGER
varying_arg	INTEGER
vector_byte_signed	FILL(x) where *x* is length[1]
vector_byte_unsigned	FILL(x) where *x* is length[1]
vector_longword_signed	FILL(4*x) where *x* is length[1]
vector_longword_unsigned	FILL(4*x) where *x* is length[1]
vector_quadword_signed	FILL(8*x) where *x* is length[1]
vector_quadword_unsigned	FILL(8*x) where *x* is length[1]
vector_word_signed	FILL(2*x) where *x* is length[1]
vector_word_unsigned	FILL(2*x) where *x* is length[1]
word_signed	FILL(2)[1]
word_unsigned	FILL(2)[1]

[1]FILL is a data type that can always be used. A FILL is an object between 0 and 65K bytes in length. VAX SCAN does not interpret the contents of an object. Thus, it can be used to pass or return the object to another language that does understand the type.

Index

How to Order Additional Documentation

Technical Support

If you need help deciding which documentation best meets your needs, call 800-343-4040 before placing your electronic, telephone, or direct mail order.

Electronic Orders

To place an order at the Electronic Store, dial 800-DEC-DEMO (800-332-3366) using a 1200- or 2400-baud modem. If you need assistance using the Electronic Store, call 800-DIGITAL (800-344-4825).

Telephone and Direct Mail Orders

Your Location	Call	Contact
Continental USA, Alaska, or Hawaii	800-DIGITAL	Digital Equipment Corporation P.O. Box CS2008 Nashua, New Hampshire 03061
Puerto Rico	809-754-7575	Local Digital subsidiary
Canada	800-267-6215	Digital Equipment of Canada Attn: DECdirect Operations KAO2/2 P.O. Box 13000 100 Herzberg Road Kanata, Ontario, Canada K2K 2A6
International	———	Local Digital subsidiary or approved distributor
Internal[1]	———	USASSB Order Processing - WMO/E15 *or* U.S. Area Software Supply Business Digital Equipment Corporation Westminster, Massachusetts 01473

[1]For internal orders, you must submit an Internal Software Order Form (EN-01740-07).

Reader's Comments

Introduction to VMS System Routines
AA–LA66B–TE

Please use this postage-paid form to comment on this manual. If you require a written reply to a software problem and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Thank you for your assistance.

I rate this manual's:	Excellent	Good	Fair	Poor
Accuracy (software works as manual says)	☐	☐	☐	☐
Completeness (enough information)	☐	☐	☐	☐
Clarity (easy to understand)	☐	☐	☐	☐
Organization (structure of subject matter)	☐	☐	☐	☐
Figures (useful)	☐	☐	☐	☐
Examples (useful)	☐	☐	☐	☐
Index (ability to find topic)	☐	☐	☐	☐
Page layout (easy to find information)	☐	☐	☐	☐

I would like to see more/less ______________________________

____

____

What I like best about this manual is ______________________________

____

____

What I like least about this manual is ______________________________

____

____

I found the following errors in this manual:

Page Description

______ ______________________________

______ ______________________________

______ ______________________________

______ ______________________________

______ ______________________________

Additional comments or suggestions to improve this manual:

____

____

____

____

I am using **Version** ______ of the software this manual describes.

Name/Title ______________________________ Dept. ______________

Company ______________________________ Date ______________

Mailing Address ______________________________

______________________________ Phone ______________

-- Do Not Tear - Fold Here and Tape

digital™

No Postage
Necessary
if Mailed
in the
United States

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO. 33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

DIGITAL EQUIPMENT CORPORATION
Corporate User Publications—Spit Brook
ZK01-3/J35 110 SPIT BROOK ROAD
NASHUA, NH 03062-9987

-- Do Not Tear - Fold Here

Cut Along Dotted Line