OpenVMS Calling Standard

Order Number: AA–QSBBD–TE

April 2001

This standard defines the requirements, mechanisms, and conventions used in the OpenVMS interface that supports procedure-to-procedure calls for Alpha and VAX environments. The standard defines the run-time data structures, constants, algorithms, conventions, methods, and functional interfaces that enable a 32-bit or 64-bit native user-mode procedure to operate correctly in a multilanguage and multithreaded environment on Alpha and VAX processors.

Revision/Update Information: This manual supersedes the *OpenVMS Calling Standard* for OpenVMS Alpha Version 7.1 and OpenVMS VAX Version 7.1.

Software Version: OpenVMS Alpha Version 7.3
OpenVMS VAX Version 7.3

Compaq Computer Corporation
Houston, Texas

ZK5973

The Compaq *OpenVMS* documentation set is available on CD-ROM.

This document was prepared using DECdocument, Version 3.3-1b.

Contents

4 OpenVMS Argument Data Types

5 OpenVMS Argument Descriptors

6 OpenVMS Conditions

Index

Examples

Figures

Tables

Preface

The *OpenVMS Calling Standard* defines the requirements, mechanisms, and conventions used in the OpenVMS interface that supports procedure-to-procedure calls for both Alpha and VAX environments. The standard defines the run-time data structures, constants, algorithms, conventions, methods, and functional interfaces that enable a native user-mode procedure to operate correctly in a multilanguage environment on Alpha and VAX systems. Properties of the run-time environment that must apply at various points during program execution are also defined.

The 32-bit user mode of the OpenVMS Alpha standard provides a high degree of compatibility with current programs written for the OpenVMS VAX environment.

The 64-bit user mode of the OpenVMS Alpha standard is a compatible superset of the OpenVMS Alpha 32-bit environment.

The interfaces, methods, and conventions specified in this manual are primarily intended for use by implementers of compilers, debuggers, and other run-time tools, run-time libraries, and base operating systems. These specifications may or may not be appropriate for use by higher level system software and applications.

This standard is under engineering change order (ECO) control. This manual includes all ECOs through ECO #48. ECOs are approved by Compaq's Calling Standard committee.

Intended Audience

This manual primarily defines requirements for compiler and debugger writers, but the information can apply to procedure calling for all programmers in various levels of programming.

Document Structure

This manual contains six chapters. Some chapters are restricted to either an Alpha or a VAX environment.

Chapter 1 provides an overview of the standard, defines goals, and defines terms used in the text.

Chapter 2 describes the primary conventions in calling a procedure in an OpenVMS VAX environment. It defines the VAX register usage and argument-passing list as well as vector and scalar processor synchronization.

Chapter 3 describes the fundamental concepts and conventions in calling a procedure in an OpenVMS Alpha environment. The chapter identifies the Alpha register usage and addressing, and focuses on aspects of the calling standard that pertain to procedure-to-procedure flow of control.

Chapter 4 defines the argument-passing data types used in calling a procedure for both OpenVMS Alpha and OpenVMS VAX environments.

Chapter 5 defines the argument descriptors used in calling a procedure for both OpenVMS Alpha and OpenVMS VAX environments.

Chapter 6 describes the OpenVMS condition- and exception-handling requirements for both OpenVMS Alpha and OpenVMS VAX environments.

Related Documents

The following manuals contain related information:

- *VAX Architecture Reference Manual*
- *Alpha Architecture Reference Manual*
- *OpenVMS Programming Interfaces: Calling a System Routine*
- *Guide to POSIX Threads Library*
- *VAX/VMS Internals and Data Structures*
- *OpenVMS AXP Internals and Data Structures*

For additional information about Compaq *OpenVMS* products and services, access the Compaq website at the following location:

`http://www.openvms.compaq.com`

Reader's Comments

Compaq welcomes your comments on this manual. Please send comments to either of the following addresses:

Internet	**openvmsdoc@compaq.com**
Mail	Compaq Computer Corporation OSSG Documentation Group, ZKO3-4/U08 110 Spit Brook Rd. Nashua, NH 03062-2698

How To Order Additional Documentation

Use the following World Wide Web address to order additional documentation:

`http://www.openvms.compaq.com`

If you need help deciding which documentation best meets your needs, call 800-282-6672.

Conventions

The following conventions are used in this manual:

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.
PF1 *x*	A sequence such as PF1 *x* indicates that you must first press and release the key labeled PF1 and then press and release another key or a pointing device button.

Return	In examples, a key name enclosed in a box indicates that you press a key on the keyboard. (In text, a key name is not enclosed in a box.) In the HTML version of this document, this convention appears as brackets, rather than a box.
. . .	A horizontal ellipsis points in examples indicate 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 points indicate 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 command format descriptions, parentheses indicate that you must enclose choices in parentheses if you specify more than one.
[ ]	In command format descriptions, brackets indicate optional choices. You can choose one or more items or no items. Do not type the brackets on the command line. However, you must include the brackets in the syntax for OpenVMS directory specifications and for a substring specification in an assignment statement.
{ }	In command format descriptions, braces indicate required choices; you must choose at least one of the items listed. Do not type the braces on the command line.
bold text	This typeface represents the introduction of a new term or the name of an argument, an attribute, or a reason.
italic text	Italic text indicates important information, complete titles of manuals, or variables. Variables include information that varies in system output (Internal error *number*), in command lines (/PRODUCER=*name*), and in command parameters in text (where *dd* represents the predefined code for the device type).
UPPERCASE TEXT	Uppercase text indicates a command, the name of a routine, the name of a file, or the abbreviation for a system privilege.
`Monospace text`	Monospace type indicates code examples and interactive screen displays. In the C programming language, monospace type in text identifies the following elements: keywords, the names of independently compiled external functions and files, syntax summaries, and references to variables or identifiers introduced in an example.
-	A hyphen at the end of a command format description, command line, or code line indicates that the command or statement continues on the following line.
numbers	All numbers in text are assumed to be decimal unless otherwise noted. Nondecimal radixes—binary, octal, or hexadecimal—are explicitly indicated.

1
Introduction

This standard defines properties such as the run-time data structures, constants, algorithms, conventions, methods, and functional interfaces that enable a native user-mode procedure to operate correctly in a multilanguage and multithreaded environment on OpenVMS Alpha and OpenVMS VAX systems. These properties include the contents of key registers, format and contents of certain data structures, and actions that procedures must perform under certain circumstances.

This standard also defines properties of the run-time environment that must apply at various points during program execution. These properties vary in scope and applicability. Some properties apply at all points throughout the execution of standard-conforming user-mode code and must, therefore, be held constant at all times. Examples of such properties include those defined for the stack pointer and various properties of the call-chain navigation mechanism. Other properties apply only at certain points, such as call conventions that apply only at the point of transfer of control to another procedure.

Furthermore, some properties are optional depending on circumstances. For example, compilers are not obligated to follow the argument list conventions when a procedure and all of its callers are in the same module, have been analyzed by an interprocedural analyzer, or have private interfaces (such as language-support routines).

Note

In many cases, significant performance gains can be realized by selective use of nonstandard calls when the safety of such calls is known. Compiler or tools writers are encouraged to make full use of such optimizations.

The OpenVMS Alpha portion of this calling standard is intended to provide a calling standard that contains a high degree of compatibility with the OpenVMS VAX environment. Conventions that differ are, for the most part, those that are dictated by differences between the Alpha and VAX hardware architectures.

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—instructions at the call site, entry point, and returns
- Argument list—structure of the list describing the arguments to the called procedure

- Function value return—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—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 are handled in a modular fashion
- Stack unwinding—how the current thread of execution is aborted efficiently

1.1 Applicability

This standard defines the rules and conventions that govern the **native user-mode run-time environment** on Alpha and VAX processors. It is applicable to all products of Compaq Computer Corporation that execute in native user mode.

Uses of this standard include:

- All externally callable interfaces in Compaq supported, standard system software
- All intermodule calls to major software components
- All external procedure calls generated by OpenVMS language processors without interprocedural analysis or permanent private conventions (such as those used for language-support run-time library [RTL] routines)

1.2 Architectural Level

This standard defines an **implementation-level run-time software architecture** for OpenVMS operating systems.

The interfaces, methods, and conventions specified in this document are primarily intended for use by implementers of compilers, debuggers, and other run-time tools, run-time libraries, and base operating systems. These specifications may or may not be appropriate for use by higher-level system software and applications.

Compilers and run-time libraries may provide additional support of these capabilities via interfaces that are more suited for compiler and application use. This specification neither prohibits nor requires such additional interfaces.

1.3 Goals

Generally, this calling standard promotes the highest degree of performance, portability, efficiency, and consistency in the interface between called procedures of a common OpenVMS environment. Specifically, the calling standard:

- Applies to all intermodule callable interfaces in the native software system. Specifically, the standard considers the requirements of important compiled languages including Ada, BASIC, Bliss, C, C++, COBOL, FORTRAN, Pascal, LISP, PL/I, and calls to the operating system and library procedures. The needs of other languages that the OpenVMS operating system may support in the future must be met by the standard or by compatible revisions to it.

- Excludes capabilities for lower-level components (such as assembler routines) that cannot be invoked from the high-level languages.
- Allows the calling program and called procedure to be written in different languages. The standard reduces the need for using language extensions in mixed-language programs.
- Contributes to the writing of error-free, modular, and maintainable software, and promotes effective sharing and reuse of software modules.
- Provides the programmer with control over fixing, reporting, and flow of control when various types of exception conditions occur.
- Provides subsystem and application writers with the ability to override system messages toward a more suitable application-oriented interface.
- Adds no space or time overhead to procedure calls and returns that do not establish exception handlers, and minimizes time overhead for establishing handlers at the cost of increased time overhead when exceptions occur.

The OpenVMS Alpha portion of this standard:

- Supports a 32-bit user-mode environment that provides a high degree of compatibility with the OpenVMS VAX environment.
- Supports a 64-bit user-mode environment that is a compatible superset of the OpenVMS Alpha 32-bit environment.
- Simplifies coexistence with VAX procedures that execute under the translated image environment.
- Simplifies the compilation of VAX assembler source to native Alpha object code.
- Supports a multilanguage, multithreaded execution environment, including efficient, effective support for the implementation of the multithread architecture (Compaq POSIX Threads Library[1]).
- Provides an efficient mechanism for calling lightweight procedures that do not need or cannot expend the overhead of setting up a stack call frame.
- Provides for the use of a common calling sequence to invoke lightweight procedures that maintain only a register call frame and heavyweight procedures that maintain a stack call frame. This calling sequence allows a compiler to determine whether to use a stack frame based on the complexity of the procedure being compiled. A recompilation of a called routine that causes a change in stack frame usage does not require a recompilation of its callers.
- Provides condition handling, traceback, and debugging for lightweight procedures that do not have a stack frame.
- Makes efficient use of the Alpha architecture, including effectively using a larger number of registers than is contained in a conventional VAX processor.
- Minimizes the cost of procedure calls.

[1] The Compaq POSIX Threads Library was formerly known as DECthreads

The OpenVMS procedure calling mechanisms of this standard do not provide:

- Checking of argument data types, data structures, and parameter access. The VAX and Alpha protection and memory management systems do not depend on correct interactions between user-level calling and called procedures. Such extended checking might be desirable in some circumstances, but system integrity does not depend on it.
- Information for an interpretive OpenVMS Debugger. The definition of the debugger includes a debug symbol table (DST) that contains the required descriptive information.

1.4 Definitions

The following terms are used in this standard:

- **Address**: On VAX systems, a 32-bit value used to denote a position in memory. On Alpha systems, a 64-bit value used to denote a position in memory. However, many Alpha applications and user-mode facilities operate in such a manner that addresses are restricted only to values that are representable in 32 bits. This allows Alpha addresses often to be stored and manipulated as 32-bit longword values. In such cases, the 32-bit address value is always implicitly or explicitly sign-extended to form a 64-bit address for use by the Alpha hardware.
- **Argument list**: A vector of entries (longwords on VAX processors, quadwords on Alpha processors) that represents a procedure parameter list and possibly a function value.
- **Asynchronous software interrupt**: An asynchronous interruption of normal code flow caused by some software event. This interruption shares many of the properties of hardware exceptions, including forcing some out-of-line code to execute.
- **Bound procedure**: A type of procedure that requires knowledge (at run time) of a dynamically determined larger enclosing scope to function correctly.
- **Call frame**: The body of information that a procedure must save to allow it to properly return to its caller. A call frame may exist on the stack or in registers. A call frame may optionally contain additional information required by the called procedure.
- **Condition handler**: A procedure designed to handle conditions (exceptions) when they occur during the execution of a thread.
- **Condition value**: A 32-bit value (sign extended to a 64-bit value on Alpha processors) used to uniquely identify an exception condition. A condition value can be returned to a calling program as a function value or it can be signaled using the OpenVMS signaling mechanism.
- **Descriptor**: A mechanism for passing parameters where the address of a descriptor is an entry in the argument list. The descriptor contains the address of the parameter, data type, size, and additional information needed to describe fully the data passed.
- **Exception condition (or condition)**: An exceptional condition in the current hardware or software state that should be noted or fixed. Its existence causes an interrupt in program flow and forces execution of out-of-line code. Such an event might be caused by an exceptional hardware state, such as arithmetic overflows, memory access control violations, and so on, or

by actions performed by software, such as subscript range checking, assertion checking, or asynchronous notification of one thread by another.

During the time the normal control flow is interrupted by an exception, that condition is termed **active**.

- **Function**: A procedure that returns a single value in accordance with the standard conventions for value returning. Additional values are returned by means of the argument list.
- **Hardware exception**: A category of exceptions that reflect an exceptional condition in the current hardware state that should be noted or fixed by the software. Hardware exceptions can occur synchronously or asynchronously with respect to the normal program flow.
- **Immediate value**: A mechanism for passing input parameters where the actual value is provided in the argument list entry by the calling program.
- **Language-support procedure**: A procedure called implicitly to implement high-level language constructs. Such procedures are not intended to be explicitly called from user programs.
- **Library procedure**: A procedure explicitly called using the equivalent of a call statement or function reference. Such procedures are usually language independent.
- **Natural alignment**: An attribute of certain data types that refers to the placement of the data so that the lowest addressed byte of the data has an address that is a multiple of the size of the data in bytes. Natural alignment of an aggregate data type generally refers to an alignment in which all members of the aggregate are naturally aligned.

  This standard defines five natural alignments:

  - Byte—Any byte address
  - Word—Any byte address that is a multiple of 2
  - Longword—Any byte address that is a multiple of 4
  - Quadword—Any byte address that is a multiple of 8
  - Octaword—Any byte address that is a multiple of 16
- **Procedure**: A closed sequence of instructions that is entered from and returns control to the calling program.
- **Procedure value**: An address value that represents a procedure. In the VAX environment, a procedure value is the address of the entry mask that is interpreted by the CALL*x* instruction invoking the procedure. In an Alpha environment, a procedure value is the address of the procedure descriptor for the procedure.
- **Process**: An address space and at least one thread of execution. Selected security and quota checks are done on a per-process basis.

  This standard anticipates the possibility of the execution of multiple threads within a process. An operating system that provides only a single thread of execution per process is considered a special case of a multithreaded system where the maximum number of threads per process is one.
- **Reference**: A mechanism for passing parameters where the address of the parameter is provided in the argument list by the calling program.

- **Signal**: A POSIX defined concept used to cause out-of-line execution of code. (This term should not be confused with the OpenVMS usage of the word that more closely equates to *exception* as used in this document.)
- **Standard call**: Any transfer of control to a procedure by any means that presents the called procedure with the environment defined by this document and does not place additional restrictions, not defined by this document, on the called procedure.
- **Standard-conforming procedure**: A procedure that adheres to all the relevant rules set forth in this document.
- **Thread of execution (or thread)**: An entity scheduled for execution on a processor. In language terms, a thread is a computational entity used by a program unit. Such a program unit might be a task, procedure, loop, or some other unit of computation.

  All threads executing within the same process share the same address space and other process contexts, but they have a unique per-thread hardware context that includes program counter, processor status, stack pointer, and other machine registers.

  This standard applies only to threads that execute within the context of a user-mode process and are scheduled on one or more processors according to software priority. All subsequent uses of the term **thread** in this standard refer only to such user-mode process threads.
- **Thread-safe code**: Code that is compiled in such a way to ensure it will execute properly when run in a threaded environment. Thread-safe code usually adds extra instructions to do certain run-time checks and requires that thread local storage be accessed in a particular fashion.
- **Undefined**: Referring to operations or behavior for which there is no directing algorithm used across all implementations that support this standard. Such operations may be well defined for a particular implementation, but they still remain undefined with reference to this standard. The actions of undefined operations may not be required by standard-conforming procedures.
- **Unpredictable**: Referring to the results of an operation that cannot be guaranteed across all implementations of this standard. These results may be well defined for a particular implementation, but they remain unpredictable with reference to this standard. All results that are not specified in this standard, but are caused by operations defined in this standard, are considered unpredictable. A standard-conforming procedure cannot depend on unpredictable results.

2
OpenVMS VAX Conventions

This chapter describes the primary conventions in calling a procedure in an OpenVMS VAX environment.

2.1 Register Usage

In the VAX architecture, there are fifteen 32-bit-wide, general-purpose hardware registers for use with scalar and vector program operations. This section defines the rules of scalar and vector register usage.

2.1.1 Scalar Register Usage

This standard defines several general-purpose VAX registers and their scalar use, as listed in Table 2–1.

Table 2–1 VAX Register Usage

Register	Use
PC	Program counter.
SP	Stack pointer.
FP	Current stack frame pointer. This register 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 4.3.
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 CALLS or CALLG 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.

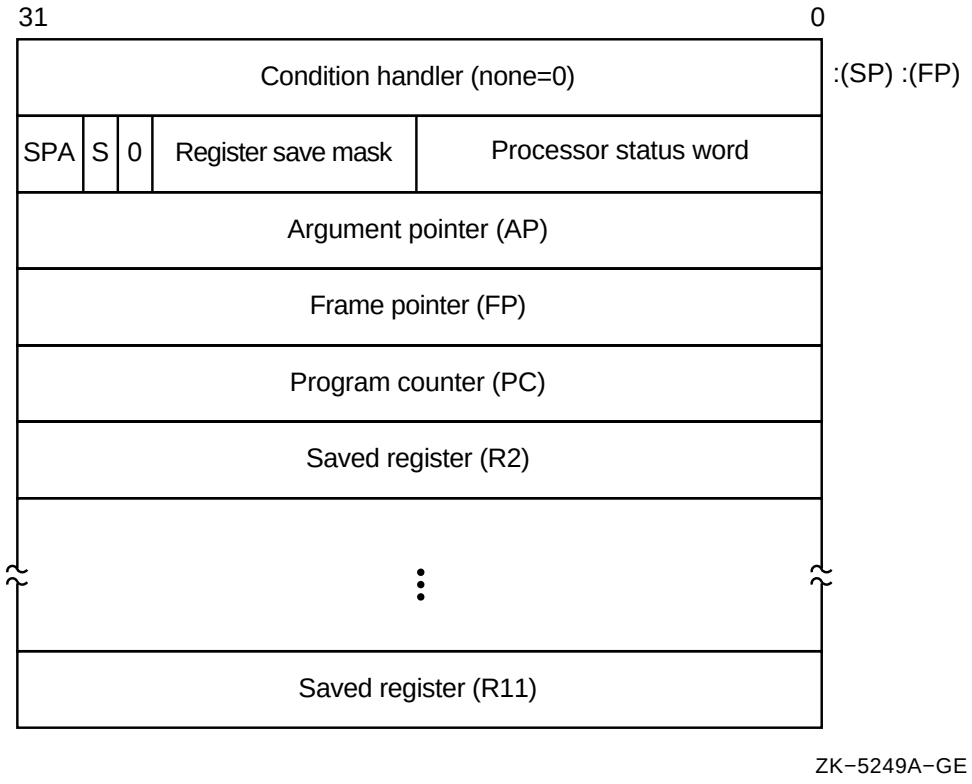
2.1.2 Vector Register Usage

This calling standard does not specify 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 preserves or otherwise manipulates the vector registers by agreement with its callers must provide an exception handler to restore them during an unwind.

2.2 Stack Usage

Figure 2–1 shows the contents of the stack frame created for the called procedure by the CALLG or CALLS instruction.

Figure 2–1 Stack Frame Generated by CALLG or CALLS Instruction



FP always points to the call frame (the condition-handler longword) of the calling procedure. Other uses of FP within a procedure are prohibited. Unless the procedure has a condition handler, the condition-handler longword contains all zeros. See Chapter 6 for more information on condition handlers.

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 return instruction (RET).

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

2.3 Calling Sequence

At the option of the calling procedure, the called procedure is invoked using 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 follows:

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

If the called procedure returns control to the calling procedure, 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 procedure by executing the RET instruction.

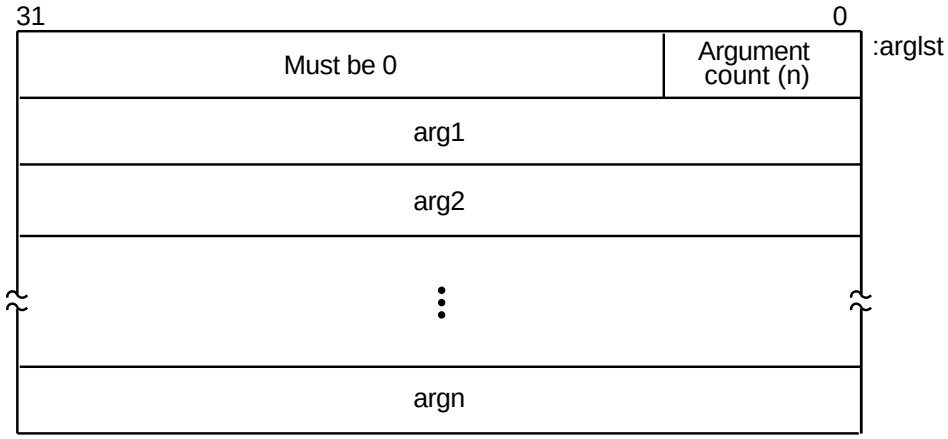
2.4 Argument List

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

2.4.1 Format

Figure 2–2 shows the argument list format.

Figure 2–2 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 to Compaq 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, array, structure, record, or a procedure.
- An address of a descriptor (by descriptor mechanism). See Chapter 5 for descriptor formats.

The standard permits programs to call by immediate value, by reference, by descriptor, or by combinations of these mechanisms. Interpretation of each argument list entry depends on agreement between the calling and called procedures. High-level languages use the reference or descriptor mechanisms for passing input parameters. OpenVMS system services and VAX BLISS, VAX C, Compaq C, Compaq 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 and 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 might be allocated in read-only memory at the option of the calling program.

2.4.2 Argument Lists and High-Level Languages

Functional notations for procedure calls in high-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 2.5 for more information.
- Each argument position corresponds to a single VAX argument list entry. For the C and C++ languages, a floating-point argument or a record `struct` that is larger than 32 bits may be passed by value using more than one VAX argument list entry. In this case, the argument count in the argument list reflects the actual number of argument list entries rather than the number of C or C++ language arguments.

2.4.2.1 Order of Argument Evaluation

Because most high-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.

Note

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

2.4.2.2 Language Extensions for Argument Transmission

This 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.

Table 2–2 lists various argument data-type groups. In the table, the value Yes means the language processor is responsible for providing the user with explicit control of that argument-passing mechanism group.

Table 2–2 Argument-Passing Mechanisms with User Explicit Control

Data Type Group	Section	Value	Reference	Descriptor
Atomic <= 32 bits	4.1	Yes	Yes	Yes
Atomic > 32 bits	4.1	No	Yes	Yes
String	4.2	No	Yes	Yes
Miscellaneous	4.3	No[1]	No	No
Array	5	No	Yes	Yes

[1]For languages that support 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 OpenVMS system services and other software. See Section 4.3.

For example, Compaq Fortran provides the following intrinsic compile-time functions:

%VAL(arg)	By immediate value mechanism. Corresponding argument list entry is the 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 descriptor of the argument **arg** as defined in Chapter 5 and in the language.

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))
```

For more information, see the Compaq Fortran language documentation.

In other languages, you can achieve the same effect 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)
```

2.5 Function Value Returns

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 4.2).

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 high-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 OpenVMS Run-Time Library procedures. For information about dynamic strings, see Section 5.3.
- If the maximum length of a fixed-length string (see Section 5.2) or a varying string (see Section 5.8) function value is not known to the calling program, the calling program can indicate that it expects the string to be returned on top of the stack. For more information about the function value return, see Section 2.5.1.

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 6.1.

2.5.1 Returning a Function Value on Top of the Stack

If the maximum length of the function value is not known, the calling program can optionally allocate certain descriptors with the POINTER field set to 0, indicating that no space has been allocated for the value. If the called procedure finds POINTER 0, it fills in the POINTER, LENGTH, and other extent fields to describe the actual size and placement of the function value. This function value is copied to the top of the stack as control returns to the calling program.

This is an exception to the usual practice because the calling program regains control at the instruction following the CALLG or CALLS sequence with the contents of SP restored to a value different from the one it had at the beginning of its CALLG or CALLS calling sequence.

This technique applies only to the first argument in the argument list. Also, the called procedure cannot assume that the calling program expects the function value to be returned on the stack. Instead, the called procedure must check the CLASS field. If the descriptor is one that can be used to return a value on the stack, the called procedure checks the POINTER field. If POINTER is not 0, the called procedure returns the value using the semantics of the descriptor. If POINTER is 0, the called procedure fills in the POINTER and LENGTH fields and returns the value to the top of the stack.

Also, when POINTER is 0, the contents of R0 and R1 are unspecified by the called procedure. Once the called procedure fills in the POINTER field and other extent fields, the calling program may pass the descriptor as an argument to other procedures.

2.5.1.1 Returning a Fixed-Length or Varying String Function Value

If a called procedure can return its function value on the stack as a fixed-length (see Section 5.2) or varying string (see Section 5.8), the called procedure must also take the following actions (determined by the CLASS and POINTER fields of the first descriptor in the argument list):

CLASS	POINTER	Called Procedure's Action
S=1	Not 0	Copy the function value to the fixed-length area specified by the descriptor and space fill (hex 20 if ASCII) or truncate on the right. The entire area is always written according to Section 5.2.
S=1	0	Return the function value on top of the stack after filling in POINTER with the first address of the string and LENGTH with the length of the string to complete the descriptor according to Section 5.2.
VS=11	Not 0	Copy the function value to the varying area specified by the descriptor and fill in CURLEN and BODY according to Section 5.8.
VS=11	0	Return the function value on top of the stack after filling in POINTER with the address of CURLEN and MAXSTRLEN with the length of the string in bytes (same value as contents of CURLEN) according to Section 5.8.
Other	–	Error. A condition is signaled.

In both the fixed-length and varying string cases, the string is unaligned. Specifically, the function value is allocated on top of the stack with no unused bytes between the stack pointer value contained at the beginning of the CALLS or CALLG sequence and the last byte of the string.

2.6 Vector and Scalar Processor Synchronization

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

Memory synchronization with the caller of a procedure that uses the vector processor is required because scalar machine writes (to main memory) might still be 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 imply that a scalar store followed by a CALL*x* 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, a SYNC instruction, or both, prior to calling 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*x* (or RET) is sometimes appropriate. The remaining two cases, where both or neither MSYNC and SYNC are needed, are 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.

2.6.1 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 and 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 instruction must be executed between each vector load and store and each standard call to other procedures to synchronize with memory operations issued by those procedures.

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 instruction to ensure scalar and vector memory synchronization can be omitted when it can be determined for the current procedure that all possibly incomplete vector load and stores operate only on memory not accessed by other procedures.

2.6.2 Exception Synchronization

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

However, if the only exceptions that can occur are certain 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 exceptions that can occur are certain 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.

3
OpenVMS Alpha Conventions

This chapter describes the fundamental concepts and conventions for calling a procedure in an Alpha environment. The following sections identify register usage and addressing, and focus on aspects of the calling standard that pertain to procedure-to-procedure flow control.

3.1 Register Usage

The 64-bit-wide, general-purpose Alpha hardware registers divide into two groups:

- Integer
- Floating point

The first 32 general-purpose registers support integer processing and the second 32 support floating-point operations.

3.1.1 Integer Registers

This standard defines the usage of the Alpha general-purpose integer registers as listed in Table 3–1.

Table 3–1 Alpha Integer Registers

Register	Usage
R0	Function value register. In a standard call that returns a nonfloating-point function result in a register, the result must be returned in this register. In a standard call, this register may be modified by the called procedure without being saved and restored. This register is not to be preserved by any called procedure.
R1	Conventional scratch register. In a standard call, this register may be modified by the called procedure without being saved and restored. This register is not to be preserved by any called procedure.
R2–15	Conventional saved registers. If a standard-conforming procedure modifies one of these registers, it must save and restore it.
R16–21	Argument registers. In a standard call, up to six nonfloating-point items of the argument list are passed in these registers. In a standard call, these registers may be modified by the called procedure without being saved and restored.
R22–24	Conventional scratch registers. In a standard call, these registers may be modified by the called procedure without being saved and restored.

(continued on next page)

Table 3–1 (Cont.) Alpha Integer Registers

Register	Usage
R25	Argument information (AI) register. In a standard call, this register describes the argument list. (See Section 3.7.1 for a detailed description.) In a standard call, this register may be modified by the called procedure without being saved and restored.
R26	Return address (RA) register. In a standard call, the return address must be passed in this register. In a standard call, this register may be modified by the called procedure without being saved and restored.
R27	Procedure value (PV) register. In a standard call, the procedure value of the procedure being called is passed in this register. In a standard call, this register may be modified by the called procedure without being saved and restored.
R28	Volatile scratch register. The contents of this register are always unpredictable after any external transfer of control either to or from a procedure. *This applies to both standard and nonstandard calls.* This register may be used by the operating system for external call fixup, autoloading, and exit sequences.
R29	Frame pointer (FP). The contents of this register define, among other things, which procedure is considered current. Details of usage and alignment are defined in Section 3.6.
R30	Stack pointer (SP). This register contains a pointer to the top of the current operating stack. Aspects of its usage and alignment are defined by the hardware architecture. Various software aspects of its usage and alignment are defined in Section 3.7.1.
R31	ReadAsZero/Sink (RZ). Hardware defines binary 0 as a source operand and sink (no effect) as a result operand.

3.1.2 Floating-Point Registers

This standard defines the usage of the Alpha general-purpose floating-point registers as listed in Table 3–2.

Table 3–2 Alpha Floating-Point Registers

Register	Usage
F0	Floating-point function value register. In a standard call that returns a floating-point result in a register, this register is used to return the real part of the result. In a standard call, this register may be modified by the called procedure without being saved and restored.
F1	Floating-point function value register. In a standard call that returns a complex floating-point result in registers, this register is used to return the imaginary part of the result. In a standard call, this register may be modified by the called procedure without being saved and restored.
F2–9	Conventional saved registers. If a standard-conforming procedure modifies one of these registers, it must save and restore it.
F10–15	Conventional scratch registers. In a standard call, these registers may be modified by the called procedure without being saved and restored.
F16–21	Argument registers. In a standard call, up to six floating-point arguments may be passed by value in these registers. In a standard call, these registers may be modified by the called procedure without being saved and restored.

(continued on next page)

Table 3–2 (Cont.) Alpha Floating-Point Registers

Register	Usage
F22–30	Conventional scratch registers. In a standard call, these registers may be modified by the called procedure without being saved and restored.
F31	ReadAsZero/Sink. Hardware defines binary 0 as a source operand and sink (no effect) as a result operand.

3.2 Address Representation

An address is a 64-bit value used to denote a position in memory. However, for compatibility with OpenVMS VAX, many Alpha applications and user-mode facilities operate in such a manner that addresses are restricted only to values that are representable in 32 bits. This allows Alpha addresses often to be stored and manipulated as 32-bit longword values. In such cases, the 32-bit address value is always implicitly or explicitly sign extended to form a 64-bit address for use by the Alpha hardware.

3.3 Procedure Representation

One distinguishing characteristic of any calling standard is how procedures are represented. The term used to denote the value that uniquely identifies a procedure is a **procedure value**. If the value identifies a bound procedure, it is called a **bound procedure value**.

In the Alpha portion of this calling standard, *all* procedure values are defined to be the address of the data structure (a procedure descriptor) that describes that procedure. So, any procedure can be invoked by calling the address stored at offset 8 from the address represented by the procedure value.

Note that a simple (unbound) procedure value is defined as the address of that procedure's descriptor (see Section 3.4). This provides slightly different conventions than would be used if the address of the procedure's code were used as it is in many calling standards.

A bound procedure value is defined as the address of a bound procedure descriptor that provides the necessary information for the bound procedure to be called (see Section 3.7.4).

3.4 Procedure Types

This standard defines the following basic types of procedures:

- **Stack frame procedure**—Maintains its caller's context on the stack
- **Register frame procedure**—Maintains its caller's context in registers
- **Null frame procedure**—Does not establish a context and, therefore, executes in the context of its caller

A compiler can choose which type of procedure to generate based on the requirements of the procedure in question. A calling procedure does not need to know what type of procedure it is calling.

Every procedure *must* have an associated structure that describes which type of procedure it is and other procedure characteristics. This structure, called a **procedure descriptor**, is a quadword-aligned data structure that provides basic information about a procedure. This data structure is used to interpret the call

chain at any point in a thread's execution. It is typically built at compile time and usually is not accessed at run time except to support exception processing or other rarely executed code.

Read access to procedure descriptors is done through a procedure interface described in Section 3.6.2. This allows for future compatible extensions to these structures.

The purpose of defining a procedure descriptor for a procedure and making that procedure descriptor accessible to the run-time system is twofold:

- To make invocations of that procedure visible to and interpretable by facilities such as the debugger, exception-handling system, and the unwinder.
- To ensure that the context of the caller saved by the called procedure can be restored if an unwind occurs. (For a description of unwinding, see Section 6.7.)

3.4.1 Stack Frame Procedures

The stack frame of a procedure consists of a fixed part (the size of which is known at compile time) and an optional variable part. Certain optimizations can be done if the optional variable part is not present. Compilers must also recognize unusual situations, such as the following, that can effectively cause a variable part of the stack to exist:

- A called routine may use the stack as a means to return certain types of function values (see Section 3.8.7 for more information).
- A called routine that allocates stack space may take an exception in its routine prologue before it becomes current. This situation must be considered since the stack expansion happens in the context of the caller (see Section 3.7.5 for more information).

  For this reason, a fixed-stack usage version of this procedure type cannot make standard calls.

The variable-stack usage version of this type of procedure is referred to as **full function** and can make standard calls to other procedures.

3.4.2 Procedure Descriptor for Procedures with a Stack Frame

A stack frame procedure descriptor (PDSC) built by a compiler provides information about a procedure with a stack frame. The minimum size of the descriptor is 32 bytes defined by constant PDSC$K_MIN_STACK_SIZE. An optional PDSC extension in 8-byte increments supports exception-handling requirements.

The fields defined in the stack frame descriptor are illustrated in Figure 3–1 and described in Table 3–3.

Figure 3–1 Stack Frame Procedure Descriptor (PDSC)

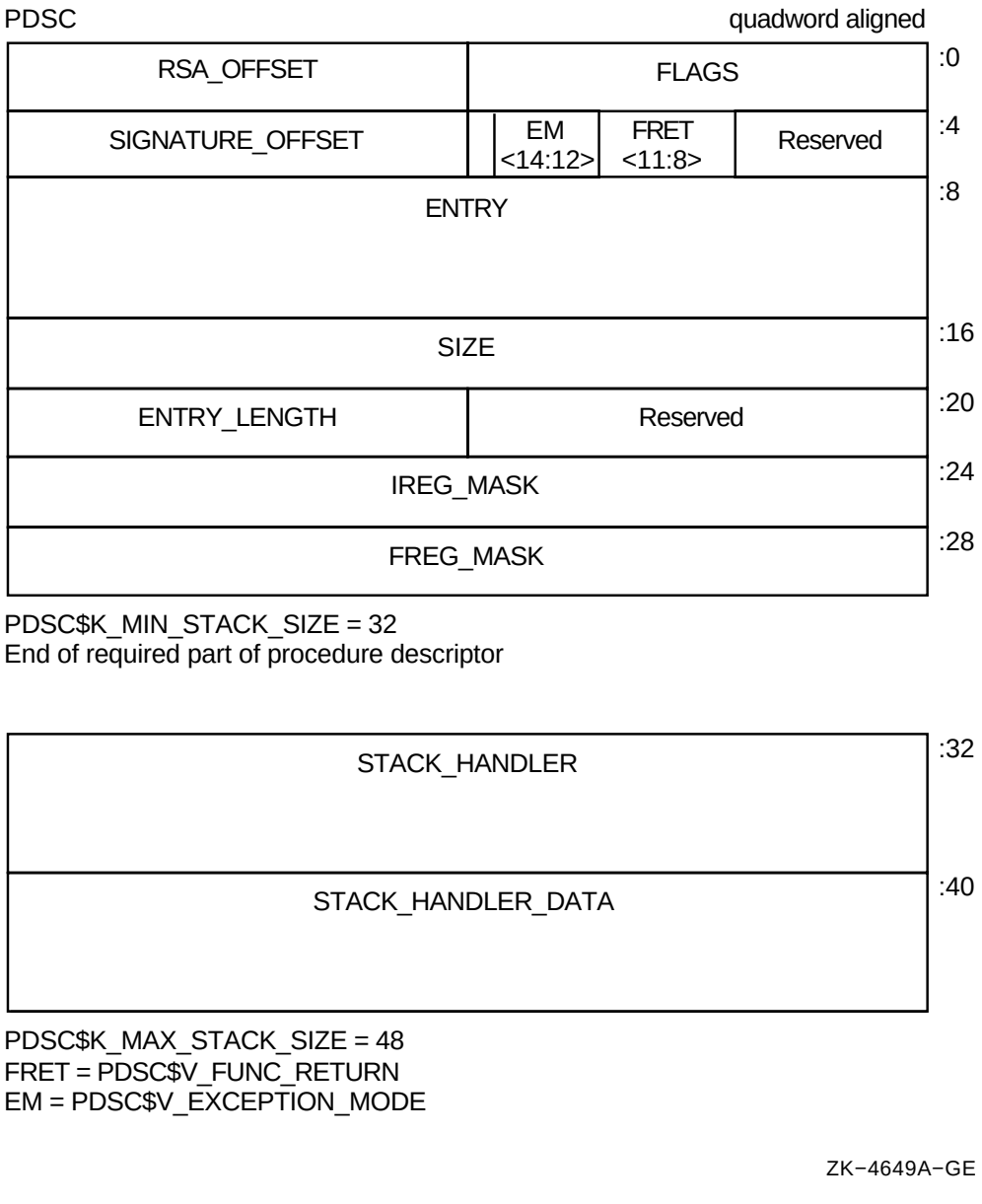


Table 3–3 Contents of Stack Frame Procedure Descriptor (PDSC)

Field Name	Contents	
PDSC$W_FLAGS	The PDSC descriptor flag bits <15:0> are defined as follows:	
	PDSC$V_KIND	A 4-bit field <3:0> that identifies the type of procedure descriptor. For a procedure with a stack frame, this field must specify a value 9 (defined by constant PDSC$K_KIND_FP_STACK).
	PDSC$V_ HANDLER_VALID	If set to 1, this descriptor has an extension for the stack handler (PDSC$Q_STACK_HANDLER) information.
	PDSC$V_ HANDLER_ REINVOKABLE	If set to 1, the handler can be reinvoked, allowing an occurrence of another exception while the handler is already active. If this bit is set to 0, the exception handler cannot be reinvoked. Note that this bit must be 0 when PDSC$V_HANDLER_VALID is 0.
	PDSC$V_ HANDLER_DATA_ VALID	If set to 1, the HANDLER_VALID bit must be 1, the PDSC extension STACK_HANDLER_DATA field contains valid data for the exception handler, and the address of PDSC$Q_STACK_HANDLER_DATA will be passed to the exception handler as defined in Section 6.2.
	PDSC$V_BASE_ REG_IS_FP	If this bit is set to 0, the SP is the base register to which PDSC$L_SIZE is added during an unwind. A fixed amount of storage is allocated in the procedure entry sequence, and SP is modified by this procedure only in the entry and exit code sequence. In this case, FP typically contains the address of the procedure descriptor for the procedure. A procedure for which this bit is 0 cannot make standard calls. If this bit is set to 1, FP is the base address and the procedure has a minimum amount of stack storage specified by PDSC$L_SIZE. A variable amount of stack storage can be allocated by modifying SP in the entry and exit code of this procedure.
	PDSC$V_REI_ RETURN	If set to 1, the procedure expects the stack at entry to be set, so an REI instruction correctly returns from the procedure. Also, if set, the contents of the RSA$Q_SAVED_RETURN field in the register save area are unpredictable and the return address is found on the stack (see Figure 3–4).
	Bit 9	Must be 0 (reserved).
	PDSC$V_BASE_ FRAME	For compiled code, this bit must be set to 0. If set to 1, this bit indicates the logical base frame of a stack that precedes all frames corresponding to user code. The interpretation and use of this frame and whether there are any predecessor frames is system software defined (and subject to change).

(continued on next page)

Table 3–3 (Cont.) Contents of Stack Frame Procedure Descriptor (PDSC)

Field Name	Contents	
	PDSC$V_TARGET_INVO	If set to 1, the exception handler for this procedure is invoked when this procedure is the target invocation of an unwind. Note that a procedure is the target invocation of an unwind if it is the procedure in which execution resumes following completion of the unwind. For more information, see Chapter 6. If set to 0, the exception handler for this procedure is not invoked. Note that when PDSC$V_HANDLER_VALID is 0, this bit must be 0.
	PDSC$V_NATIVE	For compiled code, this bit must be set to 1.
	PDSC$V_NO_JACKET	For compiled code, this bit must be set to 1.
	PDSC$V_TIE_FRAME	For compiled code, this bit must be 0. Reserved for use by system software.
	Bit 15	Must be 0 (reserved).
PDSC$W_RSA_OFFSET	Signed offset in bytes between the stack frame base (SP or FP as indicated by PDSC$V_BASE_REG_IS_FP) and the register save area. This field must be a multiple of 8, so that PDSC$W_RSA_OFFSET added to the contents of SP or FP (PDSC$V_BASE_REG_IS_FP) yields a quadword-aligned address.	
PDSC$V_FUNC_RETURN	A 4-bit field <11:8> that describes which registers are used for the function value return (if there is one) and what format is used for those registers. Table 3–7 lists and describes the possible encoded values of PDSC$V_FUNC_RETURN.	
PDSC$V_EXCEPTION_MODE	A 3-bit field <14:12> that encodes the caller's desired exception-reporting behavior when calling certain mathematically oriented library routines. The possible values for this field are defined as follows:	

Value	Name	Meaning
0	PDSC$K_EXCEPTION_MODE_SIGNAL	Raise exceptions for all error conditions except for underflows producing a 0 result. This is the default mode.
1	PDSC$K_EXCEPTION_MODE_SIGNAL_ALL	Raise exceptions for all error conditions (including underflow).
2	PDSC$K_EXCEPTION_MODE_SIGNAL_SILENT	Raise no exceptions. Create only finite values (no infinities, denormals, or NaNs). In this mode, either the function result or the C language `errno` variable must be examined for any error indication.
3	PDSC$K_EXCEPTION_MODE_FULL_IEEE	Raise no exceptions except as controlled by separate IEEE exception enable bits. Create infinities, denormals, or NaN values according to the IEEE floating-point standard.
4	PDSC$K_EXCEPTION_MODE_CALLER	Perform the exception-mode behavior specified by this procedure's caller.

(continued on next page)

Table 3–3 (Cont.) Contents of Stack Frame Procedure Descriptor (PDSC)

Field Name	Contents
PDSC$W_SIGNATURE_ OFFSET	A 16-bit signed byte offset from the start of the procedure descriptor. This offset designates the start of the procedure signature block (if any). A 0 in this field indicates that no signature information is present. Note that in a bound procedure descriptor (as described in Section 3.7.4), signature information might be present in the related procedure descriptor. A 1 in this field indicates a standard default signature. An offset value of 1 is not otherwise a valid offset because both procedure descriptors and signature blocks must be quadword aligned.
PDSC$Q_ENTRY	Absolute address of the first instruction of the entry code sequence for the procedure.
PDSC$L_SIZE	Unsigned size, in bytes, of the fixed portion of the stack frame for this procedure. The size must be a multiple of 16 bytes to maintain the minimum stack alignment required by the Alpha hardware architecture and stack alignment during a call (defined in Section 3.7.1). PDSC$L_SIZE cannot be 0 for a stack-frame type procedure, since the stack frame must include space for the register save area. The value of SP at entry to this procedure can be calculated by adding PDSC$L_SIZE to the value SP or FP, as indicated by PDSC$V_BASE_REG_ IS_FP.
PDSC$W_ENTRY_ LENGTH	Unsigned offset, in bytes, from the entry point to the first instruction in the procedure code segment following the procedure prologue (that is, following the instruction that updates FP to establish this procedure as the current procedure).
PDSC$L_IREG_MASK	Bit vector (0–31) specifying the integer registers that are saved in the register save area on entry to the procedure. The least significant bit corresponds to register R0. Never set bits 31, 30, 28, 1, and 0 of this mask, since R31 is the integer read-as-zero register, R30 is the stack pointer, R28 is always assumed to be destroyed during a procedure call or return, and R1 and R0 are never preserved registers. In this calling standard, bit 29 (corresponding to the FP) must always be set.
PDSC$L_FREG_MASK	Bit vector (0–31) specifying the floating-point registers saved in the register save area on entry to the procedure. The least significant bit corresponds to register F0. Never set bit 31 of this mask, since it corresponds to the floating-point read-as-zero register.
PDSC$Q_STACK_ HANDLER	Absolute address to the procedure descriptor for a run-time static exception-handling procedure. This part of the procedure descriptor is optional. It *must* be supplied if either PDSC$V_HANDLER_VALID is 1 or PDSC$V_ HANDLER_DATA_VALID is 1 (which requires that PDSC$V_HANDLER_ VALID be 1). If PDSC$V_HANDLER_VALID is 0, then the contents or existence of PDSC$Q_STACK_HANDLER is unpredictable.
PDSC$Q_STACK_ HANDLER_DATA	Data (quadword) for the exception handler. This is an optional quadword and needs to be supplied only if PDSC$V_HANDLER_DATA_VALID is 1. If PDSC$V_HANDLER_DATA_VALID is 0, then the contents or existence of PDSC$Q_STACK_HANDLER_DATA is unpredictable.

3.4.3 Stack Frame Format

The stack of a stack frame procedure consists of a fixed part (the size of which is known at compile time) and an optional variable part. There are two basic types of stack frames:

- Fixed size
- Variable size

Even though the exact contents of a stack frame are determined by the compiler, all stack frames have common characteristics.

Various combinations of PDSC$V_BASE_REG_IS_FP and PDSC$L_SIZE can be used as follows:

- When PDSC$V_BASE_REG_IS_FP is 0 and PDSC$L_SIZE is 0, then the procedure utilizes no stack storage and SP contains the value of SP at entry to the procedure. (Such a procedure must be a register frame procedure.)
- When PDSC$V_BASE_REG_IS_FP is 0 and PDSC$L_SIZE is a nonzero value, then the procedure has a fixed amount of stack storage specified by PDSC$L_SIZE, all of which is allocated in the procedure entry sequence, and SP is modified by this procedure only in the entry and exit code sequences. (Such a procedure may not make standard calls.)
- When PDSC$V_BASE_REG_IS_FP is 1 and PDSC$L_SIZE is a nonzero value, then the procedure has a fixed amount of stack storage specified by PDSC$L_SIZE, and may have a variable amount of stack storage allocated by modifying SP in the body of the procedure. (Such a procedure must be a stack frame procedure.)
- The combination when PDSC$V_BASE_REG_IS_FP is 1 and PDSC$L_SIZE is 0 is illegal because it violates the rules for R29 (FP) usage that requires R29 to be saved (on the stack) and restored.

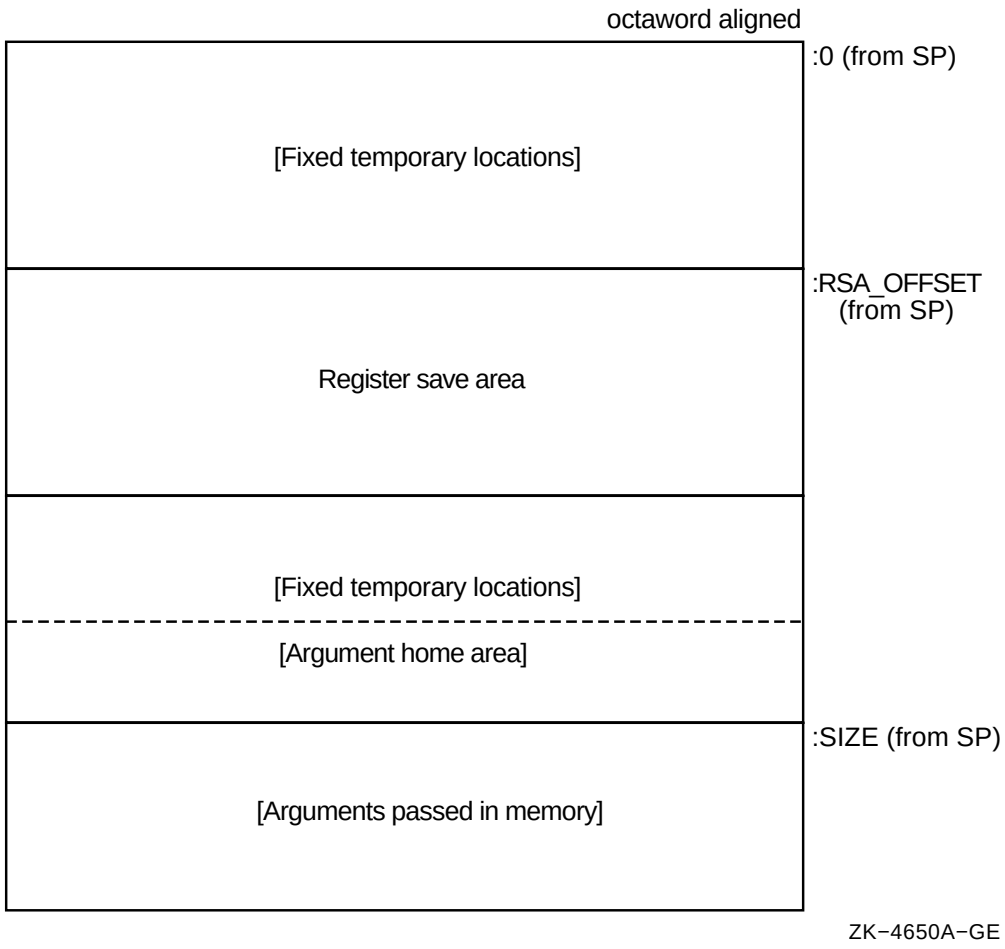
3.4.3.1 Fixed-Size Stack Frame

Figure 3–2 illustrates the format of the stack frame for a procedure with a fixed amount of stack that uses the SP register as the stack base pointer (when PDSC$V_BASE_REG_IS_FP is 0). In this case, R29 (FP) typically contains the address of the procedure descriptor for the current procedure (see Section 3.6.1).

Some parts of the stack frame are optional and occur only as required by the particular procedure. As shown in the figure, the field names within brackets are optional fields. Use of the **arguments passed in memory** field appending the end of the descriptor is described in Sections 3.4.3.3 and 3.8.2.

For information describing the fixed temporary locations and register save area, see Sections 3.4.3.3 and 3.4.3.4.

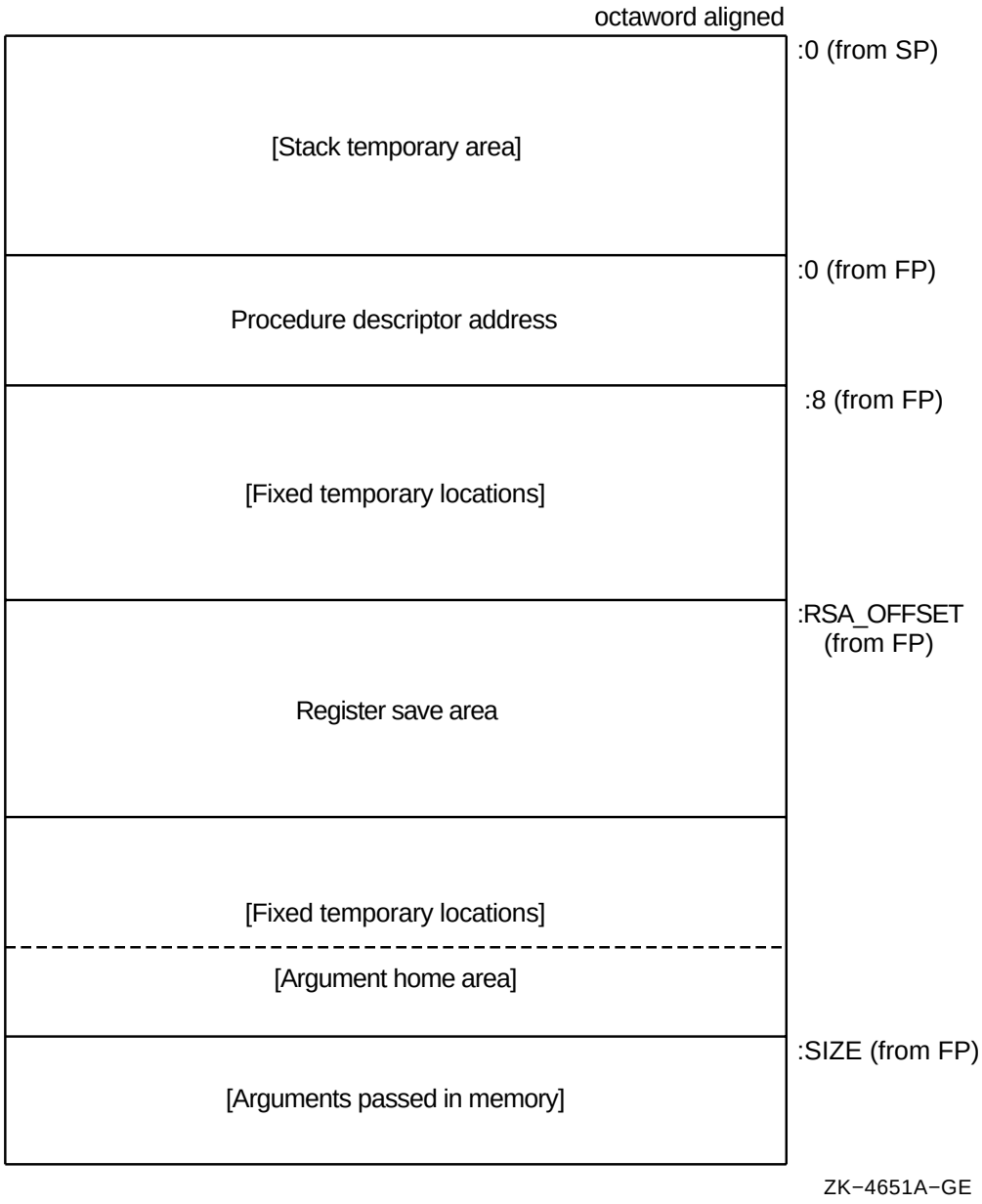
Figure 3–2 Fixed-Size Stack Frame Format



3.4.3.2 Variable-Size Stack Frame

Figure 3–3 illustrates the format of the stack frame for procedures with a varying amount of stack when PDSC$V_BASE_REG_IS_FP is 1. In this case, R29 (FP) contains the address that points to the base of the stack frame on the stack. This frame-base quadword location contains the address of the current procedure's descriptor.

Figure 3–3 Variable-Size Stack Frame Format



Some parts of the stack frame are optional and occur only as required by the particular procedure. In Figure 3–3, field names within brackets are optional fields. Use of the **arguments passed in memory** field appending the end of the descriptor is described in Sections 3.4.3.3 and 3.8.2.

For more information describing the **fixed temporary locations** and **register save area**, see Sections 3.4.3.3 and 3.4.3.4.

A compiler can use the **stack temporary area** pointed to by the SP base register for fixed local variables, such as constant-sized data items and program state, as well as for dynamically sized local variables. The stack temporary area may also be used for dynamically sized items with a limited lifetime, for example, a dynamically sized function result or string concatenation that cannot be stored directly in a target variable. When a procedure uses this area, the compiler

must keep track of its base and reset SP to the base to reclaim storage used by temporaries.

3.4.3.3 Fixed Temporary Locations for All Stack Frames

The **fixed temporary locations** are optional sections of any stack frame that contain language-specific locations required by the procedure context of some high-level languages. This may include, for example, register spill area, language-specific exception-handling context (such as language-dynamic exception-handling information), fixed temporaries, and so on.

The **argument home area** (if allocated by the compiler) can be found with the PDSC$L_SIZE offset in the last fixed temporary locations at the end of the stack frame. It is adjacent to the **arguments passed in memory** area to expedite the use of arguments passed (without copying). The argument home area is a region of memory used by the called procedure for the purpose of assembling in contiguous memory the arguments passed in registers, adjacent to the arguments passed in memory, so all arguments can be addressed as a contiguous array. This area can also be used to store arguments passed in registers if an address for such an argument must be generated. Generally, 6 * 8 bytes of stack storage is allocated for this purpose by the called procedure.

If a procedure needs to reference its arguments as a longword array or construct a structure that looks like an in-memory longword argument list, then it might allocate enough longwords in this area to hold all of the argument list and, optionally, an argument count. In that case, argument items passed in memory must be copied to this longword array.

The high-address end of the stack frame is defined by the value stored in PDSC$L_SIZE plus the contents of SP or FP, as indicated by PDSC$V_BASE_ REG_IS_FP. The high-address end is used to determine the value of SP for the predecessor procedure in the calling chain.

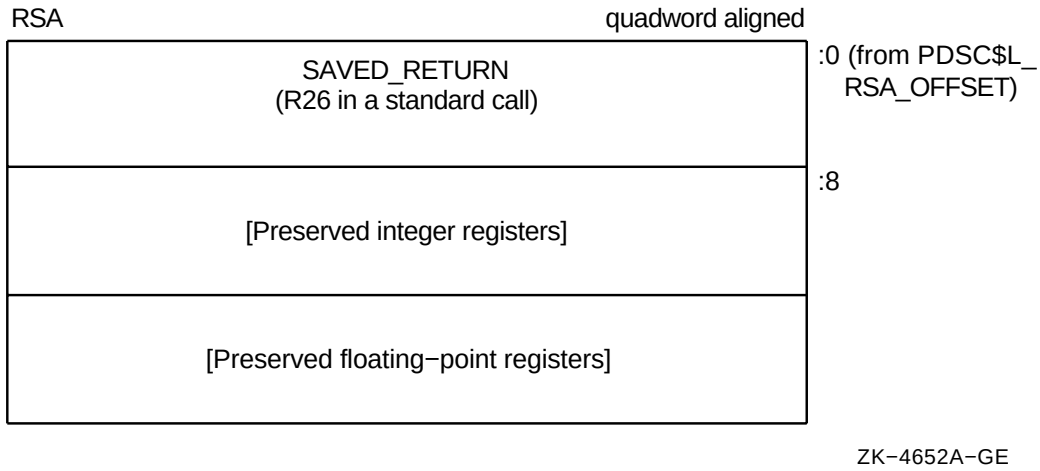
3.4.3.4 Register Save Area for All Stack Frames

The **register save area** is a set of consecutive quadwords in which registers saved and restored by the current procedure are stored (see Figure 3–4). The register save area begins at the location pointed to by the offset PDSC$W_RSA_ OFFSET from the frame base register (SP or FP as indicated by PDSC$V_BASE_ REG_IS_FP), which must yield a quadword-aligned address. The set of registers saved in this area contain the return address followed by the registers specified in the procedure descriptor by PDSC$L_IREG_MASK and PDSC$L_FREG_MASK.

All registers saved in the register save area (other than the saved return address) *must* have the corresponding bit set in the appropriate procedure descriptor register save mask even if the register is not a member of the set of registers required to be saved across a standard call. Failure to do so will prevent the correct calculation of offsets within the save area.

Figure 3–4 illustrates the fields in the register save area (field names within brackets are optional fields). Quadword RSA$Q_SAVED_RETURN is the first field in the save area and it contains the contents of the return address register. The optional fields vary in size (8-byte increments) to preserve, as required, the contents of the integer and floating-point hardware registers used in the procedure.

Figure 3–4 Register Save Area (RSA) Layout



The algorithm for packing saved registers in the quadword-aligned register save area is:

1. The return address is saved at the lowest address of the register save area (offset 0).
2. All saved integer registers (as indicated by the corresponding bit in PDSC$L_IREG_MASK being set to 1) are stored, in register-number order, in consecutive quadwords, beginning at offset 8 of the register save area.
3. All saved floating-point registers (as indicated by the corresponding bit in PDSC$L_FREG_MASK being set to 1) are stored, in register-number order, in consecutive quadwords, following the saved integer registers.

Note

Floating-point registers saved in the register save area are stored as a 64-bit exact image of the register (for example, no reordering of bits is done on the way to or from memory). Compilers must use an STT instruction to store the register regardless of floating-point type.

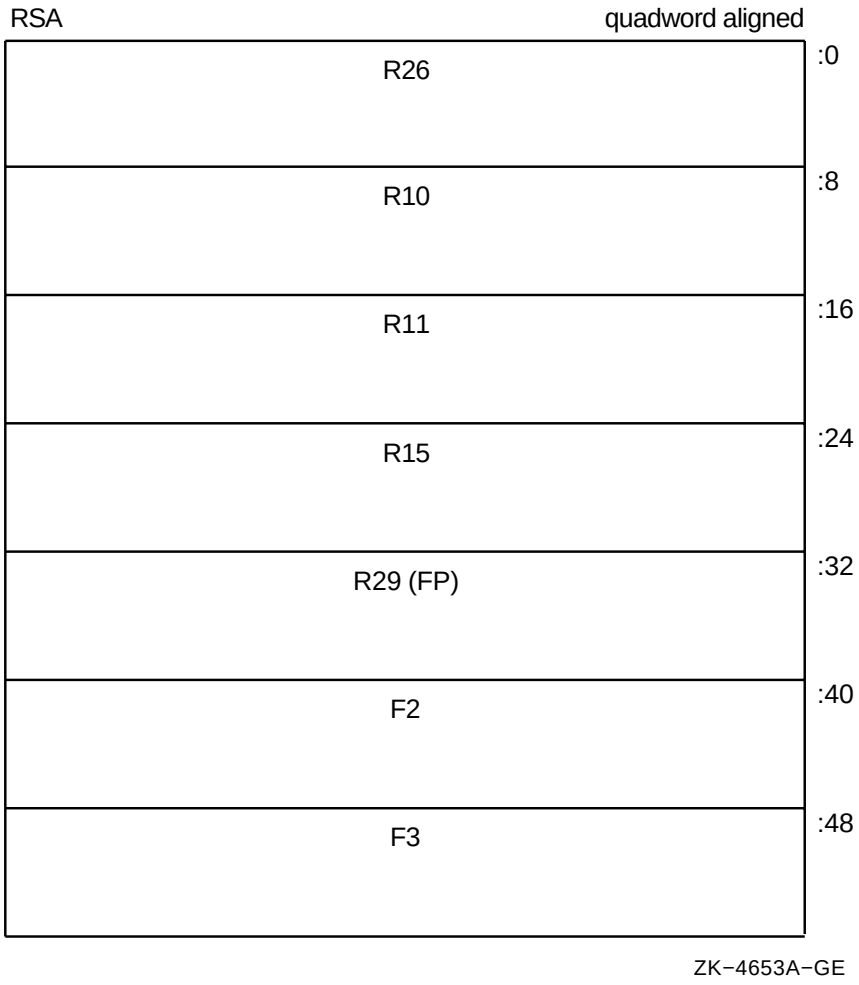
The preserved register set must *always* include R29 (FP), since it will always be used.

If the return address register is not to be preserved (as is the case for a standard call), then it must be stored at offset 0 in the register save area and the corresponding bit in the register save mask must *not* be set.

However, if a nonstandard call is made that requires the return address register to be saved and restored, then it must be stored in *both* the location at offset 0 in the register save area and at the appropriate location within the variable part of the save area. In addition, the appropriate bit of PDSC$L_IREG_MASK must be set to 1.

The example register save area shown in Figure 3–5 illustrates the register packing when registers R10, R11, R15, FP, F2, and F3 are being saved for a procedure called with a standard call.

Figure 3–5 Register Save Area (RSA) Example



3.4.4 Register Frame Procedure

A **register frame procedure** does not maintain a call frame on the stack and must, therefore, save its caller's context in registers. This type of procedure is sometimes referred to as a **lightweight procedure**, referring to the expedient way of saving the call context.

Such a procedure cannot save and restore nonscratch registers. Because a procedure without a stack frame must use scratch registers to maintain the caller's context, such a procedure cannot make a standard call to any other procedure.

A procedure with a register frame can have an exception handler and can handle exceptions in the normal way. Such a procedure can also allocate local stack storage in the normal way, although it might not necessarily do so.

Note

Lightweight procedures have more freedom than might be apparent. By using appropriate agreements with callers of the lightweight procedure, with procedures that the lightweight procedure calls, and by the use of unwind handlers, a lightweight procedure can modify nonscratch registers and can call other procedures.

Such agreements may be by convention (as in the case of language-support routines in the RTL) or by interprocedural analysis. However, calls employing such agreements are *not* standard calls and might not be fully supported by a debugger; for example, the debugger might not be able to find the contents of the preserved registers.

Since such agreements must be permanent (for upwards compatibility of object code), lightweight procedures should, in general, follow the normal restrictions.

3.4.5 Procedure Descriptor for Procedures with a Register Frame

A **register frame procedure descriptor** built by a compiler provides information about a procedure with a register frame. The minimum size of the descriptor is 24 bytes (defined by PDSC$K_MIN_REGISTER_SIZE). An optional PDSC extension in 8-byte increments supports exception-handling requirements.

The fields defined in the register frame procedure descriptor are illustrated in Figure 3–6 and described in Table 3–4.

Figure 3–6 Register Frame Procedure Descriptor (PDSC)

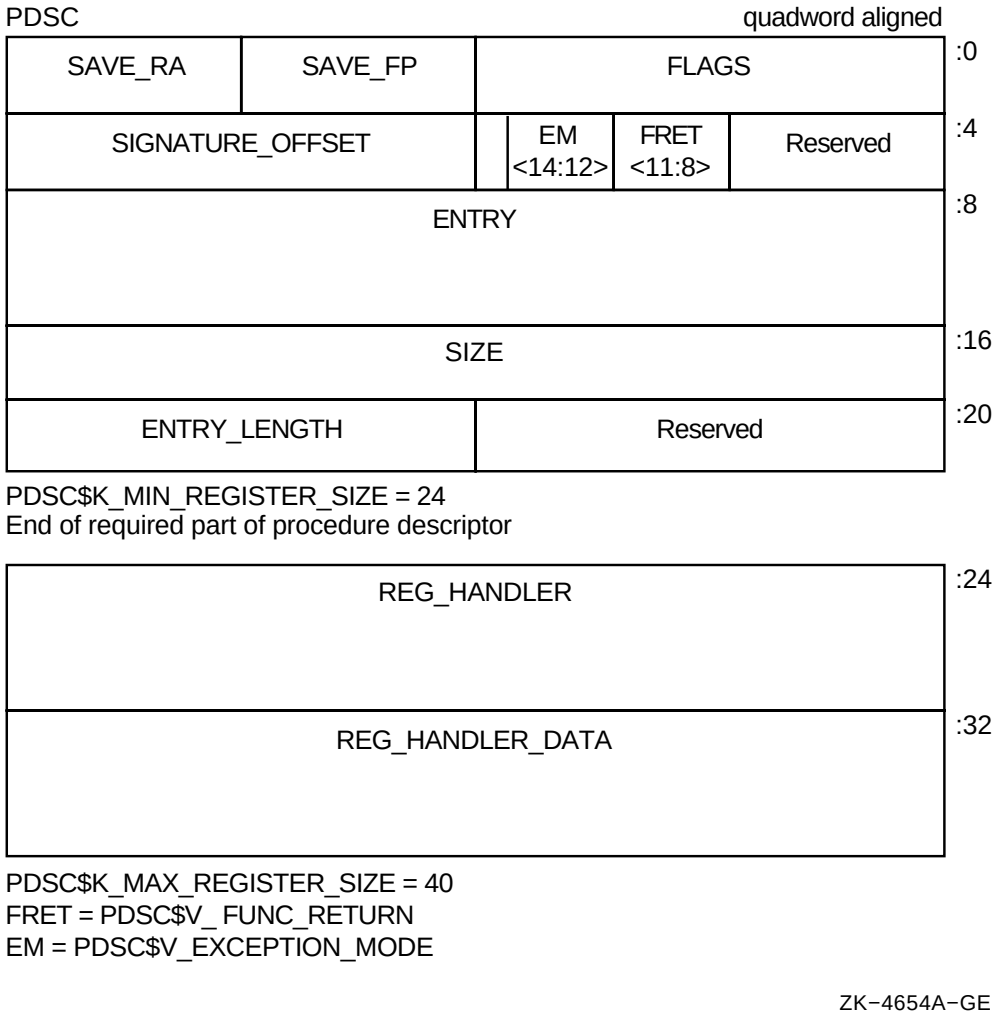


Table 3–4 Contents of Register Frame Procedure Descriptor (PDSC)

Field Name	Contents	
PDSC$W_FLAGS	The PDSC descriptor flag bits <15:0> are defined as follows:	
	PDSC$V_KIND	A 4-bit field <3:0> that identifies the type of procedure descriptor. For a procedure with a register frame, this field must specify a value 10 (defined by constant PDSC$K_KIND_FP_REGISTER).
	PDSC$V_ HANDLER_VALID	If set to 1, this descriptor has an extension for the stack handler (PDSC$Q_REG_HANDLER) information.
	PDSC$V_ HANDLER_ REINVOKABLE	If set to 1, the handler can be reinvoked, allowing an occurrence of another exception while the handler is already active. If this bit is set to 0, the exception handler cannot be reinvoked. This bit must be 0 when PDSC$V_HANDLER_VALID is 0.
	PDSC$V_ HANDLER_DATA_ VALID	If set to 1, the HANDLER_VALID bit must be 1 and the PDSC extension STACK_HANDLER_DATA field contains valid data for the exception handler, and the address of PDSC$Q_STACK_HANDLER_DATA will be passed to the exception handler as defined in Section 6.2.
	PDSC$V_BASE_ REG_IS_FP	If this bit is set to 0, the SP is the base register to which PDSC$L_SIZE is added during an unwind. A fixed amount of storage is allocated in the procedure entry sequence, and SP is modified by this procedure only in the entry and exit code sequence. In this case, FP typically contains the address of the procedure descriptor for the procedure. Note that a procedure that sets this bit to 0 cannot make standard calls. If this bit is set to 1, FP is the base address and the procedure has a fixed amount of stack storage specified by PDSC$L_SIZE. A variable amount of stack storage can be allocated by modifying SP in the entry and exit code of this procedure.
	PDSC$V_REI_ RETURN	If set to 1, the procedure expects the stack at entry to be set, so an REI instruction correctly returns from the procedure. Also, if set, the contents of the PDSC$B_SAVE_RA field are unpredictable and the return address is found on the stack.
	Bit 9	Must be 0 (reserved).
	PDSC$V_BASE_ FRAME	For compiled code, this bit must be 0. If set to 1, this bit indicates the logical base frame of a stack that precedes all frames corresponding to user code. The interpretation and use of this frame and whether there are any predecessor frames is system software defined (and subject to change).

(continued on next page)

Table 3–4 (Cont.) Contents of Register Frame Procedure Descriptor (PDSC)

Field Name	Contents	
	PDSC$V_TARGET_ INVO	If set to 1, the exception handler for this procedure is invoked when this procedure is the target invocation of an unwind. Note that a procedure is the target invocation of an unwind if it is the procedure in which execution resumes following completion of the unwind. For more information, see Chapter 6. If set to 0, the exception handler for this procedure is not invoked. Note that when PDSC$V_HANDLER_ VALID is 0, this bit must be 0.
	PDSC$V_NATIVE	For compiled code, this bit must be set to 1.
	PDSC$V_NO_ JACKET	For compiled code, this bit must be set to 1.
	PDSC$V_TIE_ FRAME	For compiled code, this bit must be 0. Reserved for use by system software.
	Bit 15	Must be 0 (reserved).
PDSC$B_SAVE_FP	Specifies the number of the register that contains the saved value of the frame pointer (FP) register. In a standard procedure, this field must specify a scratch register so as not to violate the rules for procedure entry code as specified in Section 3.7.5.	
PDSC$B_SAVE_RA	Specifies the number of the register that contains the return address. If this procedure uses standard call conventions and does not modify R26, then this field can specify R26. In a standard procedure, this field must specify a scratch register so as not to violate the rules for procedure entry code as specified in Section 3.7.5.	
PDSC$V_FUNC_ RETURN	A 4-bit field <11:8> that describes which registers are used for the function value return (if there is one) and what format is used for those registers. Table 3–7 lists and describes the possible encoded values of PDSC$V_FUNC_ RETURN.	

(continued on next page)

Table 3–4 (Cont.) Contents of Register Frame Procedure Descriptor (PDSC)

Field Name	Contents
PDSC$V_EXCEPTION_MODE	A 3-bit field <14:12> that encodes the caller's desired exception-reporting behavior when calling certain mathematically oriented library routines. The possible values for this field are defined as follows:

Value	Name	Meaning
0	PDSC$K_EXCEPTION_MODE_SIGNAL	Raise exceptions for all error conditions except for underflows producing a 0 result. This is the default mode.
1	PDSC$K_EXCEPTION_MODE_SIGNAL_ALL	Raise exceptions for all error conditions (including underflows).
2	PDSC$K_EXCEPTION_MODE_SIGNAL_SILENT	Raise no exceptions. Create only finite values (no infinities, denormals, or NaNs). In this mode, either the function result or the C language `errno` variable must be examined for any error indication.
3	PDSC$K_EXCEPTION_MODE_FULL_IEEE	Raise no exceptions except as controlled by separate IEEE exception enable bits. Create infinities, denormals, or NaN values according to the IEEE floating-point standard.
4	PDSC$K_EXCEPTION_MODE_CALLER	Perform the exception-mode behavior specified by this procedure's caller.

PDSC$W_SIGNATURE_OFFSET	A 16-bit signed byte offset from the start of the procedure descriptor. This offset designates the start of the procedure signature block (if any). A 0 in this field indicates no signature information is present. Note that in a bound procedure descriptor (as described in Section 3.7.4), signature information might be present in the related procedure descriptor. A 1 in this field indicates a standard default signature. An offset value of 1 is not otherwise a valid offset because both procedure descriptors and signature blocks must be quadword aligned.
PDSC$Q_ENTRY	Absolute address of the first instruction of the entry code sequence for the procedure.
PDSC$L_SIZE	Unsigned size in bytes of the fixed portion of the stack frame for this procedure. The size must be a multiple of 16 bytes to maintain the minimum stack alignment required by the Alpha hardware architecture and stack alignment during a call (defined in Section 3.7.1).
PDSC$W_ENTRY_LENGTH	Unsigned offset in bytes from the entry point to the first instruction in the procedure code segment following the procedure prologue (that is, following the instruction that updates FP to establish this procedure as the current procedure).
PDSC$Q_REG_HANDLER	Absolute address to the procedure descriptor for a run-time static exception-handling procedure. This part of the procedure descriptor is optional. It *must* be supplied if either PDSC$V_HANDLER_VALID is 1 or PDSC$V_HANDLER_DATA_VALID is 1 (which requires that PDSC$V_HANDLER_VALID be 1). If PDSC$V_HANDLER_VALID is 0, then the contents or existence of PDSC$Q_REG_HANDLER is unpredictable.

(continued on next page)

Table 3–4 (Cont.) Contents of Register Frame Procedure Descriptor (PDSC)

Field Name	Contents
PDSC$Q_REG_ HANDLER_DATA	Data (quadword) for the exception handler. This is an optional quadword and needs to be supplied only if PDSC$V_HANDLER_DATA_VALID is 1. If PDSC$V_HANDLER_DATA_VALID is 0, then the contents or existence of PDSC$Q_REG_HANDLER_DATA is unpredictable.

3.4.6 Null Frame Procedures

A procedure may conform to this standard even if it does not establish its own context if, in *all* circumstances, invocations of that procedure do not need to be visible or debuggable. This is termed **executing in the context of the caller** and is similar in concept to a conventional VAX JSB procedure. For the purposes of stack tracing or unwinding, such a procedure is never considered to be current.

For example, if a procedure does not establish an exception handler or does not save and restore registers, and does not extend the stack, then that procedure might not need to establish a context. Likewise, if that procedure does extend the stack, it still might not need to establish a context if the immediate caller either cannot be the target of an unwind or is prepared to reset the stack if it is the target of an unwind.

The circumstances under which procedures can run in the context of the caller are complex and are not fully specified by this standard.

As with the other procedure types previously described, the choice of whether to establish a context belongs to the called procedure. By defining a null procedure descriptor format, the same invocation code sequence can be used by the caller for all procedure types.

3.4.7 Procedure Descriptor for Null Frame Procedures

The **null frame procedure descriptor** built by a compiler provides information about a procedure with no frame. The size of the descriptor is 16 bytes (defined by PDSC$K_NULL_SIZE).

The fields defined in the null frame descriptor are illustrated in Figure 3–7 and described in Table 3–5.

Figure 3–7 Null Frame Procedure Descriptor (PDSC) Format

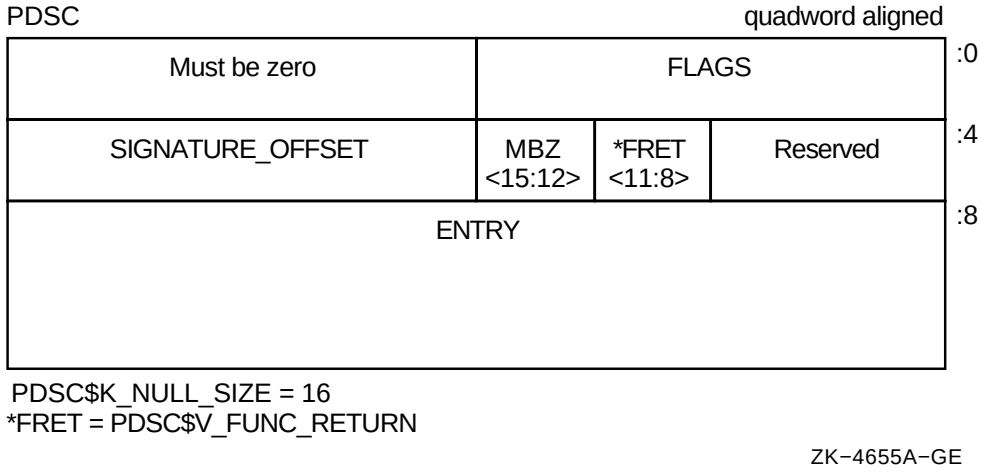


Table 3–5 Contents of Null Frame Procedure Descriptor (PDSC)

Field Name	Contents	
PDSC$W_FLAGS	The PDSC descriptor flag bits <15:0> are defined as follows:	
	PDSC$V_KIND	A 4-bit field <3:0> that identifies the type of procedure descriptor. For a null frame procedure, this field must specify a value 8 (defined by constant PDSC$K_ KIND_NULL).
	Bits 4–7	Must be 0.
	PDSC$V_REI_ RETURN	Bit 8. If set to 1, the procedure expects the stack at entry to be set, so an REI instruction correctly returns from the procedure. Also, if set, the contents of the PDSC$B_SAVE_RA field are unpredictable and the return address is found on the stack.
	Bit 9	Must be 0 (reserved).
	PDSC$V_BASE_ FRAME	For compiled code, this bit must be 0. If set to 1, indicates the logical base frame of a stack that precedes all frames corresponding to user code. The interpretation and use of this frame and whether there are any predecessor frames is system software defined (and subject to change).
	Bit 11	Must be 0 (reserved).
	PDSC$V_NATIVE	For compiled code, this bit must be set to 1.
	PDSC$V_NO_ JACKET	For compiled code, this bit must be set to 1.
	PDSC$V_TIE_ FRAME	For compiled code, this bit must be 0. Reserved for use by system software.
	Bit 15	Must be 0 (reserved).
PDSC$V_FUNC_ RETURN	A 4-bit field <11:8> that describes which registers are used for the function value return (if there is one) and what format is used for those registers.	
	Table 3–7 lists and describes the possible encoded values of PDSC$V_FUNC_ RETURN.	

(continued on next page)

Table 3–5 (Cont.) Contents of Null Frame Procedure Descriptor (PDSC)

Field Name	Contents
PDSC$W_SIGNATURE_ OFFSET	A 16-bit signed byte offset from the start of the procedure descriptor. This offset designates the start of the procedure signature block (if any). A 0 in this field indicates that no signature information is present. Note that in a bound procedure descriptor (as described in Section 3.7.4), signature information might be present in the related procedure descriptor. A 1 in this field indicates a standard default signature. An offset value of 1 is not otherwise a valid offset because both procedure descriptors and signature blocks must be quadword aligned.
PDSC$Q_ENTRY	The absolute address of the first instruction of the entry code sequence for the procedure.

3.5 Procedure Signatures

As a way to enhance certain aspects of program interoperation between images built from native Alpha code and images translated from VAX code, native Alpha compilers can optionally generate information that describes the parameters of a procedure. This auxiliary information is called procedure signature information, or sometimes just **signature information**.

Signature information is used when a call from a native procedure passes control to a translated procedure and vice versa. Translated VAX code on Alpha processors uses a VAX argument list and function return conventions as described in Sections 2.4 and 2.5. Here, the signature information is used to control how passed and returned arguments according to Alpha conventions are manipulated and placed for use by translated VAX code and vice versa.

If a procedure is compiled with signature information, PDSC$W_SIGNATURE_ OFFSET contains a byte offset from the procedure descriptor to the start of a **procedure signature control block**. The maximum size of the procedure signature control block is 72 bytes (defined by constant PSIG$K_MAX_SIZE). The fields defined in the procedure signature information block are illustrated in Figure 3–8 and described in Table 3–6.

Figure 3–8 Procedure Signature Information Block (PSIG)

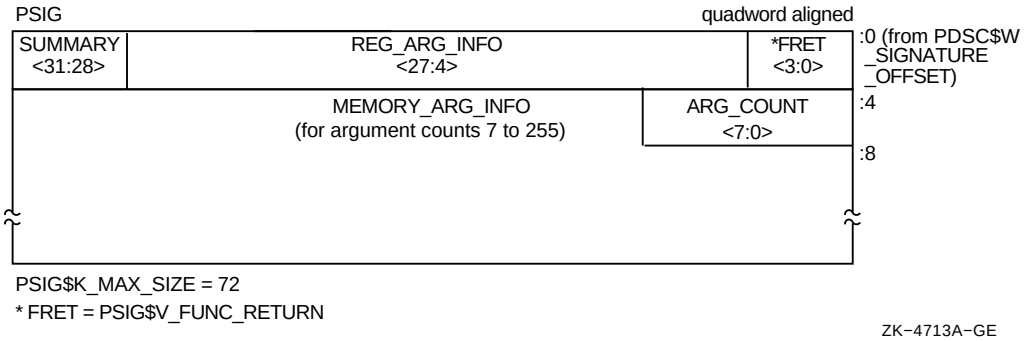


Table 3–6 Contents of the Procedure Signature Information Block (PSIG)

Field Name	Contents
PSIG$V_FUNC_RETURN	A 4-bit field <3:0> that describes which registers are used for the function value return (if there is one) and what format is used for those registers. Table 3–7 lists and describes the possible encoded values of PSIG$V_FUNC_RETURN.
PSIG$V_REG_ARG_INFO	A 24-bit field <27:4> that is divided into six groups of 4 bits that correspond to the six arguments that can be passed in registers. These groups describe how each of the first six arguments are to be passed in registers of the first group (bits <7:4>) describing the first argument. Each register argument signature group is encoded as follows:

Value	Name	Meaning[1,2]
0	RASE$K_RA_NOARG	Argument is not present
1	RASE$K_RA_Q	64-bit argument passed in an integer register
2	RASE$K_RA_I32	32-bit argument sign extended to 64 bits passed in an integer register
3	RASE$K_RA_U32	32-bit unsigned argument zero extended to 64 bits passed in an integer register
4	RASE$K_RA_FF	F_floating argument passed in a floating-point register
5	RASE$K_RA_FD	D_floating argument passed in a floating-point register
6	RASE$K_RA_FG	G_floating argument passed in a floating-point register
7	RASE$K_RA_FS	S_floating argument passed in a floating-point register
8	RASE$K_RA_FT	T_floating argument passed in a floating-point register
9–15		Reserved for future use

[1] For more specific impact on the converted field value, see Section 3.5.1.

[2] The X_floating and X_floating complex data types do not appear in this table because these types are not passed using the by value mechanism (see Section 3.8.5.1).

(continued on next page)

Table 3–6 (Cont.) Contents of the Procedure Signature Information Block (PSIG)

Field Name	Contents
PSIG$V_SUMMARY	A 4-bit field <31:28> that contains coded argument signature information as follows:

Bit	Name	Meaning
0, 1	PSIG$M_SU_ASM	Summary of arguments 7 through PSIG$B_ARG_COUNT: 00 = All arguments are 64-bit or not used 01 = All arguments are 32-bit sign extended or not used 10 = Reserved 11 = Other (not 00 or 01)
2	PSIG$M_SU_VLIST	VAX formatted argument list expected
3		Must be 0 (reserved)

PSIG$M_SU_ASM values of 00 and 01 (binary) allow a quick test for the occurrence of either an all 32-bit or an all 64-bit argument list. The values for the PSIG$V_MEMORY_ARG_INFO field must be valid even when these occurrences apply.

PSIG$B_ARG_COUNT — Unsigned byte (bits 0–7) that specifies the number of 64-bit argument items described in the argument signature information. This count includes the first six arguments.

PSIG$V_MEMORY_ARG_INFO — Array of 2-bit values that describe each of arguments 7 through PSIG$B_ARG_COUNT. PSIG$S_MEMORY_ARG_INFO data is only defined for the arguments described by PSIG$B_ARG_COUNT. These memory argument signature bits are defined as follows:

Value	Name	Meaning[1]
0	MASE$K_MA_Q	64-bit argument
1		Reserved
2	MASE$K_MA_I32	32-bit sign-extended argument
3		Reserved

[1]For more specific impact on the converted field value, see Section 3.5.1.

Table 3–7 Function Return Signature Encodings

Value	Name	Meaning[1,2]
0	PSIG$K_FR_I64	64-bit result in R0 *or* No function result provided *or* First parameter mechanism used
1	PSIG$K_FR_D64	64-bit result with low 32 bits sign extended in R0 and high 32 bits sign extended in R1
2	PSIG$K_FR_I32	32-bit sign extended to 64-bit result in R0

[1]For more specific impact on the converted field value, see Section 3.5.1.

[2]The X_floating and X_floating complex data types do not appear in this table because these types are not passed using the by value mechanism (see Section 3.8.5.1).

(continued on next page)

Table 3–7 (Cont.) Function Return Signature Encodings

Value	Name	Meaning[1,2]
3	PSIG$K_FR_U32	32-bit unsigned result (zero extended) in R0
4	PSIG$K_FR_FF	F_floating result in F0
5	PSIG$K_FR_FD	D_floating result in F0
6	PSIG$K_FR_FG	G_floating result in F0
7	PSIG$K_FR_FS	S_floating result in F0
8	PSIG$K_FR_FT	T_floating result in F0
9, 10		Reserved for future use
11	PSIG$K_FR_FFC	F_floating complex result in F0 and F1
12	PSIG$K_FR_FDC	D_floating complex result in F0 and F1
13	PSIG$K_FR_FGC	G_floating complex result in F0 and F1
14	PSIG$K_FR_FSC	S_floating complex result in F0 and F1
15	PSIG$K_FR_FTC	T_floating complex result in F0 and F1

[1]For more specific impact on the converted field value, see Section 3.5.1.

[2]The X_floating and X_floating complex data types do not appear in this table because these types are not passed using the by value mechanism (see Section 3.8.5.1).

3.5.1 Call Parameter PSIG Conversions

Where a VAX image is translated to an Alpha image, the VAX registers R0–15 are represented using the lower half of the corresponding Alpha registers R0–15 at call interface boundaries. No “type conversion” is performed in making parameters from either native or translated code available to each other. However, it is important to understand the effects of the PSIG field values when interfacing between native and translated environments.

Note that an address under OpenVMS Alpha is described using RASE$K_RA_I32 or MASE$K_MA_I32 as appropriate.

3.5.1.1 Native-to-Translated Code PSIG Conversions

The specific impact of the native-to-translated call conversions of the PSIG$V_REG_ARG_INFO and the PSIG$V_FUNC_RETURN field values are listed in Table 3–8.

Table 3–8 Native-to-Translated Conversion of the PSIG Field Values

Name	Impact
PSIG$V_REG_ARG_INFO Field Conversions	
RASE$K_RA_Q	The low-order 32 bits of the integer register contents are used to fill the first of two longword entries in the VAX formatted argument list, while the high-order 32 bits are used to fill the second longword entry. This counts as two arguments in the VAX formatted argument list.
RASE$K_RA_I32 RASE$K_RA_U32	The low-order 32 bits of the integer register contents are used to fill one longword entry in the VAX formatted argument list passed to the translated procedure. The high-order 32 bits are ignored. This counts as one argument in the VAX formatted argument list.
RASE$K_RA_FF	The single-precision contents of a floating-point register are used to fill one longword entry in the VAX formatted argument list passed to the translated procedure. This counts as one argument in the VAX formatted argument list. The Alpha store instruction STF is used to place the register contents into memory.
RASE$K_RA_FD RASE$K_RA_FG	The double-precision contents of a floating-point register are used to fill two longword entries in the VAX formatted argument list passed to the translated procedure. This counts as two arguments in the VAX formatted argument list. The Alpha store instruction STG is used to place the register contents into memory.
RASE$K_RA_FS RASE$K_RA_FT	Undefined.
PSIG$V_MEMORY_ARG_INFO Field Conversions	
MASE$K_MA_Q MASE$K_MA_I32	These convert like the RASE$K_RA_Q and RASE$K_RA_I32 field conversions, except that the Alpha argument list entry is stored in memory (rather in than a register).
PSIG$V_FUNC_RETURN Field Conversions	
PSIG$K_FR_I64	The translated code is returning a 64-bit result split between R0 and R1. The low-order 32 bits of R1 are shifted left and combined with the low-order 32 bits of R0 to form the 64-bit result that is returned to the native caller.
PSIG$K_FR_D64	The translated code is returning a 64-bit result split between R0 and R1. Both R0 and R1 are sign extended from 32 to 64 bits and returned to the native caller in place.
PSIG$K_FR_I32 PSIG$K_FR_U32	The translated code is returning a 32-bit result in R0. R0 is sign extended from 32 to 64 bits and returned to the native caller in place.
PSIG$K_FR_FF	The single-precision contents of the result in R0 is loaded into Alpha register F0.
PSIG$K_FR_FD PSIG$K_FR_FG	The double-precision contents in registers R0 and R1 are combined and loaded into Alpha register F0.
PSIG$K_FR_FS PSIG$K_FR_FT	Undefined.

(continued on next page)

Table 3–8 (Cont.) Native-to-Translated Conversion of the PSIG Field Values

Name	Impact
PSIG$V_FUNC_RETURN Field Conversions	
PSIG$K_FR_FFC	The single-precision complex contents in registers R0 and R1 are loaded into Alpha registers F0 and F1.
PSIG$K_FR_FDC PSIG$K_FR_FGC	The translated code is returning a double-precision complex result using the hidden first parameter method (by reference). The storage for the result is allocated prior to the call and the address is passed as the extra parameter. Upon return, the result is copied from the temporary storage into the Alpha floating-point registers and returned to the native caller.
PSIG$K_FR_FSC PSIG$K_FR_FTC	Undefined.

In all 64-bit cases, the longword at the lower memory address forms the earlier argument in the VAX formatted argument list. Also, for single-precision floating-point types, the unused 32 bits of an Alpha 64-bit argument list entry are undefined.

3.5.1.2 Translated-to-Native Code PSIG Conversions

The specific impact of the translated-to-native call conversions of the PSIG$V_REG_ARG_INFO and the PSIG$V_FUNC_RETURN field values are listed in Table 3–9.

Table 3–9 Translated-to-Native Conversion of the PSIG Field Values

Name	Impact
PSIG$V_REG_ARG_INFO Field Conversions	
RASE$K_RA_Q	The contents of two successive longwords from the VAX formatted argument list are combined to form a single quadword value that is placed in an integer register. This counts as one argument in the Alpha argument list.
RASE$K_RA_I32 RASE$K_RA_U32	The contents of one longword entry from the VAX formatted argument list is sign extended and placed in the integer register. This counts as one argument in the Alpha argument list.
RASE$K_RA_FF	A single longword entry from the VAX formatted argument list is used to form a floating-point value in a floating-point register. This counts as one argument in the Alpha argument list. The Alpha load instruction LDF is used to place the argument in the floating-point register.
RASE$K_RA_FD RASE$K_RA_FG	Two longword entries from the VAX formatted argument list are combined to form a single floating-point value in a floating-point register. This counts as one argument in the Alpha argument list. The Alpha load instruction LDG is used to place the argument in the floating-point register.
RASE$K_RA_FS RASE$K_RA_FT	Undefined.

(continued on next page)

Table 3–9 (Cont.) Translated-to-Native Conversion of the PSIG Field Values

Name	Impact
PSIG$V_MEMORY_ARG_INFO Field Conversions	
MASE$K_MA_Q MASE$K_MA_I32	These convert like RASE$K_RA_Q and RASE$K_RA_I32 field conversions, except that the Alpha argument list entry is stored in memory (rather than a register).[1]
PSIG$V_FUNC_RETURN Field Conversions	
PSIG$K_FR_I64	The native code is returning a 64-bit result in R0. The high 32 bits of R0 are moved to the low half of R1 and sign extended, and then R0 is sign extended from 32 to 64 bits. The 64-bit result is then returned to the translated caller in R0 and R1.
PSIG$K_FR_D64	The native code is returning a 64-bit result split between R0 and R1. Both R0 and R1 are sign extended from 32 to 64 bits and returned to the translated caller in place.
PSIG$K_FR_I32 PSIG$K_FR_U32	The native code is returning a 32-bit result in R0. R0 is sign extended from 32 to 64 bits and the result is then returned in place to the translated caller.
PSIG$K_FR_FF	The single-precision result in Alpha register F0 is stored in the low-order half of register R0.[1]
PSIG$K_FR_FD PSIG$K_FR_FG	The double-precision result in Alpha register F0 is stored in the low-order halves of registers R0 and R1.
PSIG$K_FR_FS PSIG$K_FR_FT	Undefined.
PSIG$K_FR_FFC	The single-precision complex result in Alpha registers F0 and F1 is stored in the low-order halves of registers R0 and R1.[1]
PSIG$K_FR_FDC PSIG$K_FR_FGC	The native code is returning a double-precision complex result in the Alpha floating-point registers. The result is copied into the storage given by the hidden first parameter passed by the translated caller.
PSIG$K_FR_FSC PSIG$K_FR_FTC	Undefined.

[1]Note that for single-precision floating-point types, the unused 32 bits of an Alpha 64-bit argument list entry are undefined.

3.5.2 Default Procedure Signature

In certain cases, a standard default procedure signature representing a common combination of characteristics is encoded in a special manner. (For example, see the descriptions of PDSC$W_SIGNATURE_OFFSET in Sections 3.4.1 and 3.7.4.)

In the OpenVMS Alpha environment, procedure signatures are used only to effect interoperation between native OpenVMS Alpha and translated VAX VMS or OpenVMS VAX images. Default procedure signature characteristics are defined for each of the two possible call situations.

For an OpenVMS Alpha procedure that is callable from a translated VAX procedure, a default procedure signature implies the following characteristics about the expected parameters and result of a call to that procedure:

- The number of parameters passed is contained in the AI (R25) register (taken from the count in the VAX argument list).
- All parameters (if any) are 32-bit sign extended (RASE$K_RA_I32 for register arguments, MASE$K_MA_I32 for memory arguments).

- The function result (if any) is 32-bit sign extended (PSIG$K_FR_I32).

For a bound procedure used as a jacket to effect a call into a translated image, a default procedure signature implies the following characteristics about the actual parameters and the expected result from a call to that translated procedure:

- The number of parameters passed is contained in the AI (R25) register.
- The register parameters (if any) are described in the AI register.
- The memory parameters (if any) are 32-bit sign extended (MASE$K_MA_I32).
- The function result (if any) is 32-bit sign extended (PSIG$K_FR_I32).

3.6 Procedure Call Chain

Except for the first invocation in a thread, there is always an invocation that was previously considered to be the **current procedure** invocation. The current procedure invocation, together with the previous current procedure invocations, together with all successive previously current procedure invocations, all the way back to the first invocation in the thread, make up a logical list of procedure contexts referred to as the **call chain**. The current procedure invocation is always considered to be the first procedure invocation in this logical list and the first procedure invocation executed in the thread is always the last procedure invocation in the list. The register values of all nonscratch registers at the time of the currently active call in a procedure invocation can be determined by walking the call chain and retrieving the procedure invocation context for that invocation. A procedure is called an **active procedure** (active invocation) while it exists on the call chain.

The call chain and its supporting data are used by code that implements various aspects of the calling standard such as call returns and procedure unwinding.

3.6.1 Current Procedure

In this calling standard, R29 is the frame pointer (FP) register that defines the current procedure.

Therefore, the current procedure must *always* maintain in FP one of the following pointer values:

- Pointer to the procedure descriptor for that procedure.
- Pointer to a naturally aligned quadword containing the address of the procedure descriptor for that procedure. For purposes of finding a procedure's procedure descriptor, no assumptions must be made about the quadword location. As long as all other requirements of this standard are met, a compiler is free to use FP as a base register for any arbitrary storage, including a stack frame, provided that while the procedure is current, the quadword pointed to by the value in FP contains the address of that procedure's descriptor.

At any point in time, the FP value can be interpreted to find the procedure descriptor for the current procedure by examining the value at 0(FP) as follows:

- If 0(FP)<2:0> = 0, then FP points to a quadword that contains a pointer to the procedure descriptor for the current procedure.
- If 0(FP)<2:0> $\neq$ 0, then FP points to the procedure descriptor for the current procedure.

By examining the first quadword of the procedure descriptor, the procedure type can be determined from the PDSC$V_KIND field.

The following code is an example of how the current procedure descriptor and procedure type can be found:

```
        LDQ     R0,0(FP)        ;Fetch quadword at FP
        AND     R0,#7,R28       ;Mask alignment bits
        BNEQ    R28,20$         ;Is procedure descriptor pointer
        LDQ     R0,0(R0)        ;Was pointer to procedure descriptor
10$:    AND     R0,#7,R28       ;Do sanity check
        BNEQ    R28,20$         ;All is well

        ;Error - Invalid FP

20$:    AND     R0,#15,R0       ;Get kind bits

        ;Procedure KIND is now in R0
```

IF PDSC$V_KIND is equal to PDSC$K_KIND_FP_STACK, the current procedure has a stack frame.

If PDSC$V_KIND is equal to PDSC$K_KIND_FP_REGISTER, the current procedure is a register frame procedure.

Either type of procedure can use either type of mechanism to point to the procedure descriptor. Compilers may choose the appropriate mechanism to use based on the needs of the procedure involved.

3.6.2 Procedure Call Tracing

Mechanisms for each of the following functions are needed to support procedure call tracing:

- To provide the context of a procedure invocation
- To walk (navigate) the procedure call chain
- To refer to a given procedure invocation

This section describes the data structure mechanisms. The routines that support these functions are described in Section 3.6.3.

3.6.2.1 Referring to a Procedure Invocation from a Data Structure

When referring to a specific procedure invocation at run time, a **procedure invocation handle**, shown in Figure 3–9, can be used. Defined by constant LIBICB$K_INVO_HANDLE_SIZE, the structure is a single-field longword called HANDLE. HANDLE describes the invocation handle of the procedure.

Figure 3–9 Procedure Invocation Handle Format

To encode a procedure invocation handle, follow these steps:

1. If PDSC$V_BASE_REG_IS_FP is set to 1 in the corresponding procedure descriptor, then set INVO_HANDLE to the contents of the FP register in that invocation.

If PDSC$V_BASE_REG_IS_FP is set to 0, set INVO_HANDLE to the contents of the SP register in that invocation. (That is, start with the base register value for the frame.)

2. Shift the INVO_HANDLE contents left one bit. Because this value is initially known to be octaword aligned (see Section 3.7.1), the result is a value whose 5 low-order bits are 0.
3. If PDSC$V_KIND = PDSC$K_KIND_FP_STACK, perform a logical OR on the contents of INVO_HANDLE with the value $1F_{16}$, and then set INVO_HANDLE to the value that results.

   If PDSC$V_KIND = PDSC$K_KIND_FP_REGISTER, perform a logical OR on the contents of INVO_HANDLE with the contents of PDSC$B_SAVE_RA, and then set INVO_HANDLE to the value that results.

Note that a procedure invocation handle is not defined for a null frame procedure.

Note

So you can distinguish an invocation of a register frame procedure that calls another register frame procedure (where the called procedure uses no stack space and therefore has the same base register value as the caller), the register number that saved the return address is included in the invocation handle of a register frame procedure. Similarly, the number 31_{10} in the invocation handle of a stack frame procedure is included to distinguish an invocation of a stack frame procedure that calls a register frame procedure where the called procedure uses no stack space.

3.6.2.2 Invocation Context Block

The context of a specific procedure invocation is provided through the use of a data structure called an **invocation context block**. The minimum size of the block is 528 bytes and is system defined using the constant LIBICB$K_INVO_CONTEXT_BLK_SIZE. The size of the last field (LIBICB$Q_SYSTEM_DEFINED[*n*]) defined by the host system determines the total size of the block.

The fields defined in the invocation context block are illustrated in Figure 3–10 and described in Table 3–10.

Figure 3–10 Invocation Context Block Format

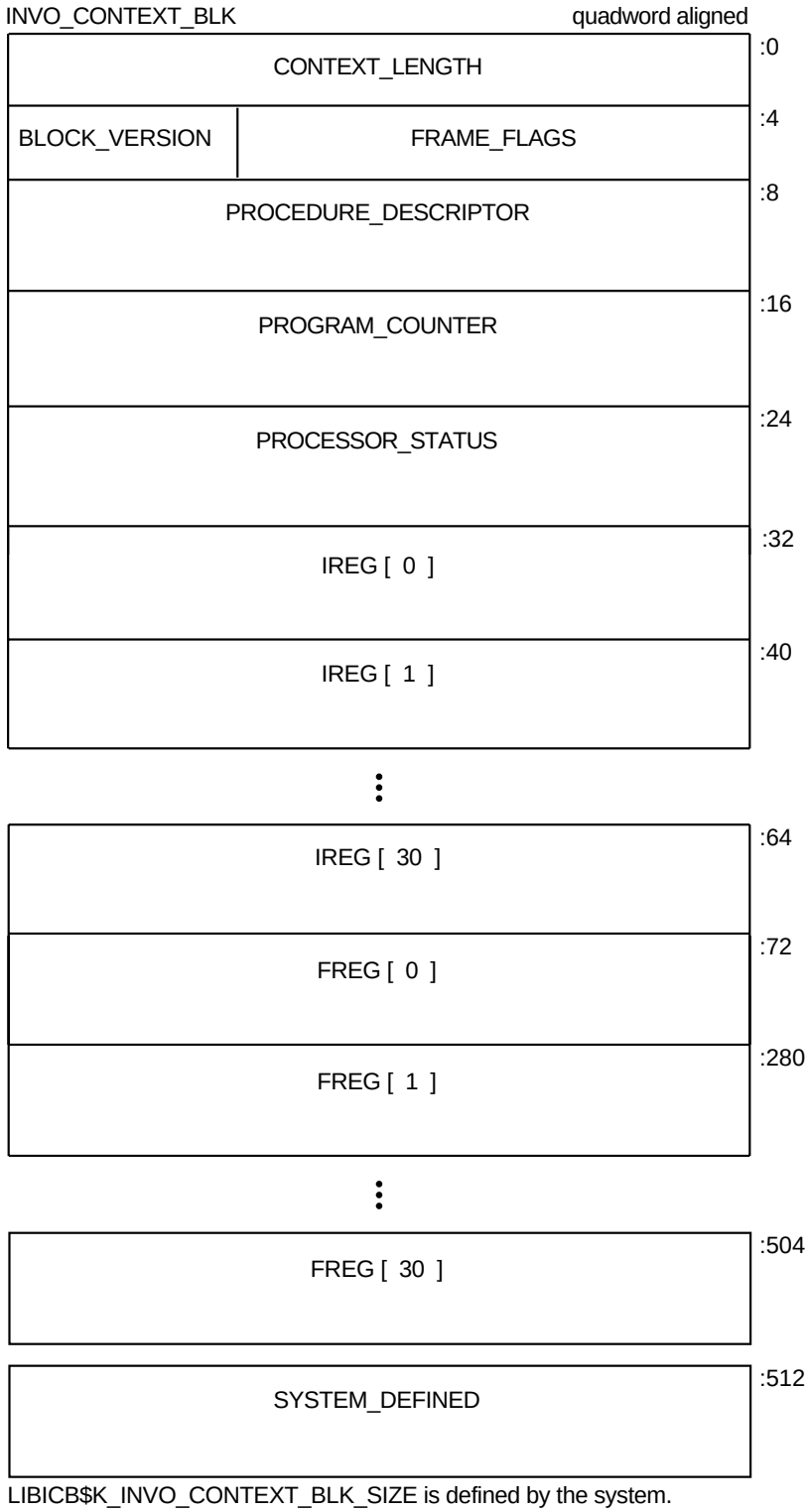


Table 3–10 Contents of the Invocation Context Block

Field Name	Contents	
LIBICB$L_CONTEXT_LENGTH	Unsigned count of the total length in bytes of the context block; this represents the sum of the lengths of the standard-defined portion and the system-defined section.	
LIBICB$R_FRAME_FLAGS	The procedure frame flag bits <24:0> are defined as follows:	
	LIBICB$V_EXCEPTION_FRAME	Bit 0. If set to 1, the invocation context corresponds to an exception frame.
	LIBICB$V_AST_FRAME	Bit 1. If set to 1, the invocation context corresponds to an asynchronous trap.
	LIBICB$V_BOTTOM_OF_STACK	Bit 2. If set to 1, the invocation context corresponds to a frame that has no predecessor.
	LIBICB$V_BASE_FRAME	Bit 3. If set to 1, the BASE_FRAME bit is set in the FLAGS field of the associated procedure descriptor.
LIBICB$B_BLOCK_VERSION	A byte that defines the version of the context block. Since this block is currently the first version, the value is set to 1.	
LIBICB$PH_PROCEDURE_DESCRIPTOR	Address of the procedure descriptor for this context.	
LIBICB$Q_PROGRAM_COUNTER	Quadword that contains the current value of the procedure's program counter. For interrupted procedures, this is the same as the continuation program counter; for active procedures, this is the return address back into that procedure.	
LIBICB$Q_PROCESSOR_STATUS	Contains the current value of the processor status.	
LIBICB$Q_IREG[*n*]	Quadword that contains the current value of the integer register in the procedure (where *n* is the number of the register).	
LIBICB$Q_FREG[*n*]	Quadword that contains the current value of the floating-point register in the procedure (where *n* is the number of the register).	
LIBICB$Q_SYSTEM_DEFINED[*n*]	A variable-sized area with locations defined in quadword increments by the host environment that contains procedure context information. These locations are *not* defined by this standard.	

3.6.2.3 Getting a Procedure Invocation Context with a Routine

A thread can obtain its own context or the current context of any procedure invocation in the current call chain (given an invocation handle) by calling the run-time library functions defined in Section 3.6.3.

3.6.2.4 Walking the Call Chain

During the course of program execution, it is sometimes necessary to walk the call chain. Frame-based exception handling is one case where this is done. Call chain navigation is possible only in the reverse direction (in a latest-to-earliest or top-to-bottom procedure).

To walk the call chain, perform the following steps:

1. Build an invocation context block when given a program state (which contains a register set).

   For the current routine, an initial invocation context block can be obtained by calling the LIB$GET_CURR_INVO_CONTEXT routine (see Section 3.6.3.2).

2. Repeatedly call the LIB$GET_PREV_INVO_CONTEXT routine (see Section 3.6.3.3) until the end of the chain has been reached (as signified by 0 being returned).

Compilers are allowed to optimize high-level language procedure calls in such a way that they do not appear in the invocation chain. For example, inline procedures never appear in the invocation chain.

Make no assumptions about the relative positions of any memory used for procedure frame information. There is no guarantee that successive stack frames will always appear at higher addresses.

3.6.3 Invocation Context Access Routines

A thread can manipulate the invocation context of any procedure in the thread's virtual address space by calling the following run-time library functions.

3.6.3.1 LIB$GET_INVO_CONTEXT

A thread can obtain the invocation context of any active procedure by using the following function format:

LIB$GET_INVO_CONTEXT(invo_handle, invo_context)

Argument	OpenVMS Usage	Type	Access	Mechanism
invo_handle	invo_handle	longword (unsigned)	read	by value
invo_context	invo_context_blk	structure	write	by reference

Arguments:

invo_handle
Handle for the desired invocation.

invo_context
Address of an invocation context block into which the procedure context of the frame specified by **invo_handle** will be written.

Function Value Returned:

status
Status value. A value of 1 indicates success; a value of 0 indicates failure.

Note

If the invocation handle that was passed does not represent any procedure context in the active call chain, the value of the new contents of the context block is unpredictable.

3.6.3.2 LIB$GET_CURR_INVO_CONTEXT

A thread can obtain the invocation context of a current procedure by using the following function format:

LIB$GET_CURR_INVO_CONTEXT(invo_context)

Argument	OpenVMS Usage	Type	Access	Mechanism
invo_context	invo_context_blk	structure	write	by reference

Argument:

invo_context
Address of an invocation context block into which the procedure context of the caller will be written.

Function Value Returned:

None. To facilitate use in the implementation of the C language unwind `setjmp` or `longjump` function (only), the routine sets R0 to 0.

3.6.3.3 LIB$GET_PREV_INVO_CONTEXT

A thread can obtain the invocation context of the procedure context preceding any other procedure context by using the following function format:

LIB$GET_PREV_INVO_CONTEXT(invo_context)

Argument	OpenVMS Usage	Type	Access	Mechanism
invo_context	invo_context_blk	structure	modify	by reference

Argument:

invo_context
Address of an invocation context block. The given context block is updated to represent the context of the previous (calling) frame.

For the purposes of this function, the minimum fields of an invocation block that must be defined are those IREG and FREG fields corresponding to registers used by a context whether the registers are preserved or not. Note that the invocation context blocks written by the routines specified in these sections define all possible fields in a context block. Such context blocks satisfy this minimum requirement.

Function Value Returned:

status
Status value. A value of 1 indicates success. When the initial context represents the bottom of the call chain, a value of 0 is returned. If the current operation completed without error, but a stack corruption was detected at the next level down, a value of 3 is returned.

3.6.3.4 LIB$GET_INVO_HANDLE

A thread can obtain an invocation handle corresponding to any invocation context block by using the following function format:

LIB$GET_INVO_HANDLE(invo_context)

Argument	OpenVMS Usage	Type	Access	Mechanism
invo_context	invo_context_blk	structure	read	by reference

Argument:

invo_context
Address of an invocation context block. Here, only the frame pointer and stack pointer fields of an invocation context block must be defined.

Function Value Returned:

invo_handle
Invocation handle of the invocation context that was passed. If the returned value is LIB$K_INVO_HANDLE_NULL, the invocation context that was passed was invalid.

3.6.3.5 LIB$GET_PREV_INVO_HANDLE

A thread can obtain an invocation handle of the procedure context preceding that of a specified procedure context by using the following function format:

LIB$GET_PREV_INVO_HANDLE(invo_handle)

Argument	OpenVMS Usage	Type	Access	Mechanism
invo_handle	invo_handle	longword (unsigned)	read	by value

Argument:

invo_handle
An invocation handle that represents a target invocation context.

Function Value Returned:

invo_handle
An invocation handle for the invocation context that is previous to that which was specified as the target.

3.6.3.6 LIB$PUT_INVO_REGISTERS

A given procedure invocation context's fields can be updated with new register contents by calling a system library function in following format:

LIB$PUT_INVO_REGISTERS(invo_handle, invo_context, invo_mask)

Argument	OpenVMS Usage	Type	Access	Mechanism
invo_handle	invo_handle	longword (unsigned)	read	by value
invo_context	invo_context_blk	structure	read	by reference
invo_mask	mask_quadword	quadword (unsigned)	read	by reference

Arguments:

invo_handle
Handle for the invocation to be updated.

invo_context
Address of an invocation context block that contains new register contents.

Each register that is set in the **invo_mask** parameter, except SP, is updated using the value found in the corresponding IREG or FREG field. The program counter and processor status can also be updated in this way. (The SP register cannot be updated using this routine.) No other fields of the invocation context block are used.

invo_mask
Address of a 64-bit bit vector, where each bit corresponds to a register field in the passed **invo_context**. Bits 0 through 30 correspond to IREG[0] through IREG[30], bit 31 corresponds to PROGRAM_COUNTER, bits 32 through 62 correspond to FREG[0] through FREG[30], and bit 63 corresponds to PROCESSOR_STATUS. (If bit 30, which corresponds to SP, is set, then no changes are made.)

Function Value Returned:

status
Status value. A value of 1 indicates success. When the initial context represents the bottom of the call chain or when bit 30 of the **invo_mask** argument is set, a value of 0 is returned (and nothing is changed).

Caution

While this routine can be used to update the frame pointer (FP), great care must be taken to assure that a valid stack frame and execution environment result; otherwise, execution may become unpredictable.

3.7 Transfer of Control

This standard states that a standard call may be accomplished in any way that presents the called routine with the required environment (see Section 1.4). However, typically, most standard-conforming external calls are implemented with a common sequence of instructions and conventions. Since a common set of call conventions is so pervasive, these conventions are included for reference as part of this standard.

One important feature of the calling standard is that the same instruction sequence can be used to call each of the different types of procedure. Specifically, the caller does not have to know which type of procedure is being called.

3.7.1 Call Conventions

The call conventions describe the rules and methods used to communicate certain information between the caller and the called procedure during invocation and return. For a standard call, these conventions include the following:

- **Procedure value**

  The calling procedure must pass to the called procedure its procedure value. This value can be a statically or dynamically bound procedure value. This is accomplished by loading R27 with the procedure value before control is transferred to the called procedure.

- **Return address**

  The calling procedure must pass to the called procedure the address to which control must be returned during a normal return from the called procedure. In most cases, the return address is the address of the instruction following the one that transferred control to the called procedure. For a standard call, this address is passed in the return address register (R26).

- **Argument list**

  The **argument list** is an ordered set of zero or more **argument items** that together constitute a logically contiguous structure known as an **argument item sequence**. This logically contiguous sequence is typically mapped to registers and memory in a way that produces a physically discontiguous argument list. In a standard call, the first six items are passed in registers R16–21 or registers F16–21. (See Section 3.8.2 for details of argument-to-register correspondence.) The remaining items are collected in a memory argument list that is a naturally aligned array of quadwords. In a standard call, this list (if present) must be passed at 0(SP).

- **Argument information**

  The calling procedure must pass to the called procedure information about the argument list. This information is passed in the argument information (AI) register (R25). Defined by AI$K_AI_SIZE, the structure is a quadword as shown in Figure 3–11 with the fields described in Table 3–11.

Figure 3–11 Argument Information Register (R25) Format

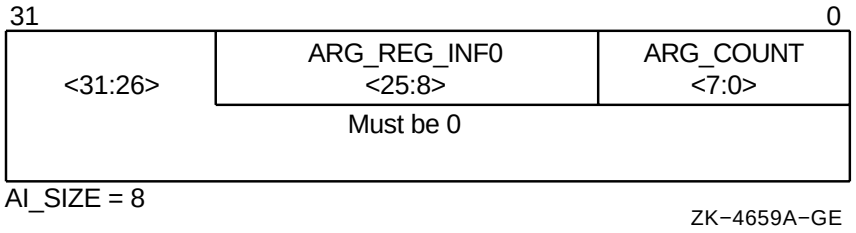


Table 3–11 Contents of the Argument Information Register (R25)

Field Name	Contents
AI$B_ARG_COUNT	Unsigned byte <7:0> that specifies the number of 64-bit argument items in the argument list (known as the "argument count").
AI$V_ARG_REG_INFO	An 18-bit vector field <25:8> divided into six groups of 3 bits that correspond to the six arguments passed in registers. These groups describe how each of the first six arguments are passed in registers with the first group <10:8> describing the first argument. The encoding for each group for the argument register usage follows:

Value	Name	Meaning
0	AI$K_AR_I64	64-bit or 32-bit sign-extended to 64-bit argument passed in an integer register *or* Argument is not present
1	AI$K_AR_FF	F_floating argument passed in a floating register
2	AI$K_AR_FD	D_floating argument passed in a floating register
3	AI$K_AR_FG	G_floating argument passed in a floating register
4	AI$K_AR_FS	S_floating argument passed in a floating register
5	AI$K_AR_FT	T_floating argument passed in a floating register
6, 7		Reserved

Bits 26–63	Reserved and must be 0.

- **Function result**

  If a standard-conforming procedure is a function and the function result is returned in a register, then the result is returned in R0, F0, or F0 and F1. Otherwise, the function result is returned via the first argument item or dynamically as defined in Section 3.8.7.

- **Stack usage**

  At any time, the stack pointer (SP) must denote an address that has the minimum alignment required by the Alpha hardware. In addition, whenever control is transferred to another procedure, the stack must be octaword aligned. (A side effect of this is that the in-memory portion of the argument list will start on an octaword boundary.) During a procedure invocation, the SP (R30) can never be set to a value higher than the value of SP at entry to that procedure invocation.

  The contents of the stack located above the portion of the argument list that is passed in memory (if any) belongs to the calling procedure and is, therefore, not to be read or written by the called procedure, except as specified by indirect arguments or language-controlled up-level references.

  Since SP is used by the hardware in raising exceptions and asynchronous interrupts, the contents of the next 2048 bytes below the current SP value are continually and unpredictably modified. Software that conforms to this standard must not depend on the contents of the 2048 stack locations below 0(SP).

  Note

  One implication of the stack alignment requirement is that low-level interrupt and exception-fielding software must be prepared to handle and correct the alignment before calling handler routines, in case the stack pointer is not octaword aligned at the time of an interrupt or exception.

3.7.2 Linkage Section

Because the Alpha hardware architecture has the property of instructions that cannot contain full virtual addresses, it is sometimes referred to as a **base register architecture**. In a base register architecture, normal memory references within a limited range from a given address are expressed by using displacements relative to the contents of a register containing that address (base register). Base registers for external program segments, either data or code, are usually loaded indirectly through a program segment of address constants.

The fundamental program section containing address constants that a procedure uses to access other static storage, external procedures, and variables is termed a **linkage section**. Any register used to access the contents of the linkage section is termed a **linkage pointer**.

A procedure's linkage section includes the procedure descriptor for the procedure, addresses of all external variables and procedures referenced by the procedure, and other constants a compiler may choose to reference using a linkage pointer.

When a standard procedure is called, the caller must provide the procedure value for that procedure in R27. Static procedure values are defined to be the address of the procedure's descriptor. Since the procedure descriptor is part of the linkage section, calling this type of procedure value provides a pointer into the linkage section for that procedure in R27. This linkage pointer can then be used by the called procedure as a base register to address locations in its linkage section. For this reason, most compilers generate references to items in the linkage section as offsets from a pointer to the procedure's descriptor.

Compilers usually arrange (as part of the environment setup) to have the environment setup code (for bound procedures) load R27 with the address of the procedure's descriptor so it can be used as a linkage pointer as previously described. For an example, see Section 3.7.4.

Although not required, linkages to external procedures are typically represented in the calling procedure's linkage section as a **linkage pair**. As shown in Figure 3–12 and described in Table 3–12, a linkage pair (LKP) block with two fields should be octaword aligned and defined by LKP$K_SIZE as 16 bytes.

Figure 3–12 Linkage Pair Block Format

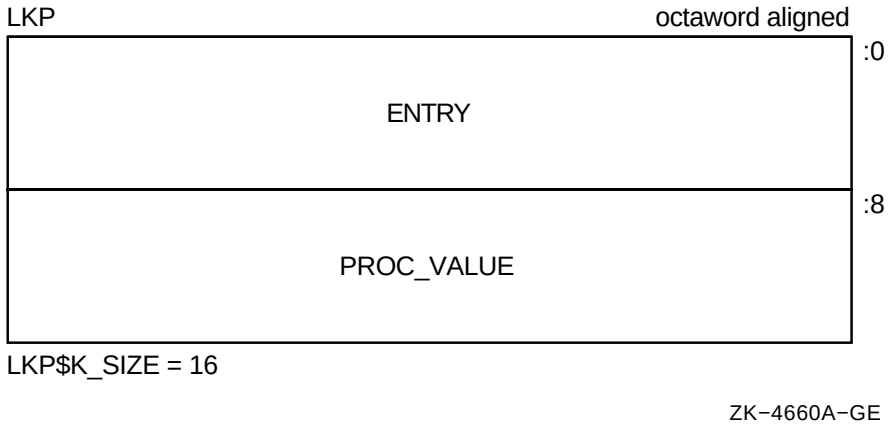


Table 3–12 Contents of the Linkage Pair Block

Field Name	Contents
LKP$Q_ENTRY	Absolute address of the first instruction of the called procedure's entry code sequence.
LKP$Q_PROC_VALUE	Contains the procedure value of the procedure to be called. Normally, this field is the absolute address of a procedure descriptor for the procedure to be called, but in certain cases, it could be a bound procedure value (such as for procedures that are called through certain types of transfer vectors).

In general, an object module contains a procedure descriptor for each entry point in the module. The descriptors are allocated in a linkage section. For each external procedure Q that is referenced in a module, the module's linkage section also contains a linkage pair denoting Q (which is a pointer to Q's procedure descriptor and entry code address).

The following code example calls an external procedure Q as represented by a linkage pair. In this example, R4 is the register that currently contains the address of the current procedure's descriptor.

```
LDQ     R26,Q_DESC-MY_DESC(R4)   ;Q's entry address into R26
LDQ     R27,Q_DESC-MY_DESC+8(R4) ;Q's procedure value into R27
MOVQ    #AI_LITERAL,R25          ;Load Argument Information register
JSR     R26,(R26)                ;Call to Q. Return address in R26
```

Because Q's procedure descriptor (statically defined procedure value) is in Q's linkage section, Q can use the value in R27 as a base address for accessing data in its linkage section. Q accesses external procedures and data in other program sections through pointers in its linkage section. Therefore, R27 serves as the root pointer through which all data can be referenced.

3.7.3 Calling Computed Addresses

Most calls are made to a fixed address whose value is determined by the time the program starts execution. However, certain cases are possible that cause the exact address to be unknown until the code is finally executed. In this case, the procedure value representing the procedure to be called is computed in a register.

The following code example illustrates a call to a computed procedure value (simple or bound) that is contained in R4:

```
        LDQ     R26,8(R4)           ;Entry address to scratch register
        MOV     R4,R27              ;Procedure value to R27
        MOVQ    #AI_LITERAL,R25     ;Load Argument Information register
        JSR     R26,(R26)           ;Call entry address.
```

If interoperation with translated images must be considered, the procedure value (in this example, in R4) might be the address of a VAX entry point rather than the address of an Alpha procedure descriptor. A VAX entry point can be dynamically distinguished from an Alpha procedure descriptor by examining bits 12 and 13 of a VAX entry call mask, which are required to be 0 by the VAX architecture. For an Alpha procedure, bit 12 corresponds to the PDSC$V_NATIVE flag, which is required to be set in all Alpha procedure descriptors. Bit 13 corresponds to the PDSC$V_NO_JACKET flag, which is currently required to be set but reserved for enhancements to this standard in all Alpha procedure descriptors.

If the procedure value is determined to correspond to an Alpha procedure, then the call can be completed as discussed. If the procedure value is determined to correspond to a VAX procedure, then the call must be completed using system facilities that will effect the transition into and out of the code of the translated image. Example 3–1 illustrates a code sequence for examining the procedure value.

Example 3–1 Code for Examining the Procedure Value

```
        LDL     R28,0(R4)             ;Load the flags field of the target PDSC
        MOVQ    #AI_LITERAL,R25       ;Load Argument Information register
        SRL     R28,#PDSC$V_NO_JACKET,R26 ;Position jacket flag
        BLBC    R26,CALL_JACKET       ;If clear then jacket needed
        LDQ     R26,8(R4)             ;Entry address to scratch register
        MOV     R4,R27                ;Procedure value to R27
        JSR     R26,(R26)             ;Call entry address.
back_in_line:
        ...                           ;Rest of procedure code goes here

TRANSLATED:                           ;Generated out of line, R2 contains a
        LDQ     R26,N_TO_T_LKP(R2)    ;Entry address to scratch register
        LDQ     R27,N_TO_T_LKP+8(R2)  ;Load procedure value
        MOV     R4,R23                ;Address of routine to call to R23
        JSR     R26,(R26)             ;Call jacket routine
        BR      back_in_line          ;Return to normal code path
```

(continued on next page)

Example 3–1 (Cont.) Code for Examining the Procedure Value

```
CALL_JACKET:                            ;
        SRL     R28,#PDSC$V_NATIVE,R28;Jacketing for translated or native?
        LDA     R24,PSIG_OUT(R2)       ;Pass address of our argument
                                       ; signature information in R24
        BLBC    R28,TRANSLATED         ;If clear, then translated jacketing
        (Native Jacketing Reserved for Future Use)
        BR      back_in_line           ;Return to normal code path
```

In Example 3–1, jacketing functionality is provided by the SYS$NATIVE_TO_TRANSLATED routine. This system procedure is called with the actual arguments for the target procedure in their normal locations (as though the target procedure were an Alpha procedure) and with two additional, nonstandard arguments in registers R23 and R24. R23 contains the procedure value for the target VAX procedure, and R24 contains the address of a procedure signature block for this call as described in Section 3.5.

3.7.4 Bound Procedure Descriptors

Bound procedure descriptors provide a mechanism to interpose special processing between a call and the called routine without modifying either. The descriptor may contain (or reference) data used as part of that processing. Between native and translated images, the OpenVMS Alpha operating system uses linker and image-activator created bound procedure descriptors to mediate the handling of parameter and result passing (see Section 3.5). Language processors on OpenVMS Alpha systems use bound procedure descriptors to implement bound procedure values (see Section 3.7.4.1). Other uses are possible.

The minimum size of the descriptor is 24 bytes (defined by PDSC$K_MIN_BOUND_SIZE). An optional PDSC extension in 8-byte increments provides the specific environment values as defined by the implementation.

The fields defined in the bound procedure descriptor are illustrated in Figure 3–13 and described in Table 3–13.

Figure 3–13 Bound Procedure Descriptor (PDSC)

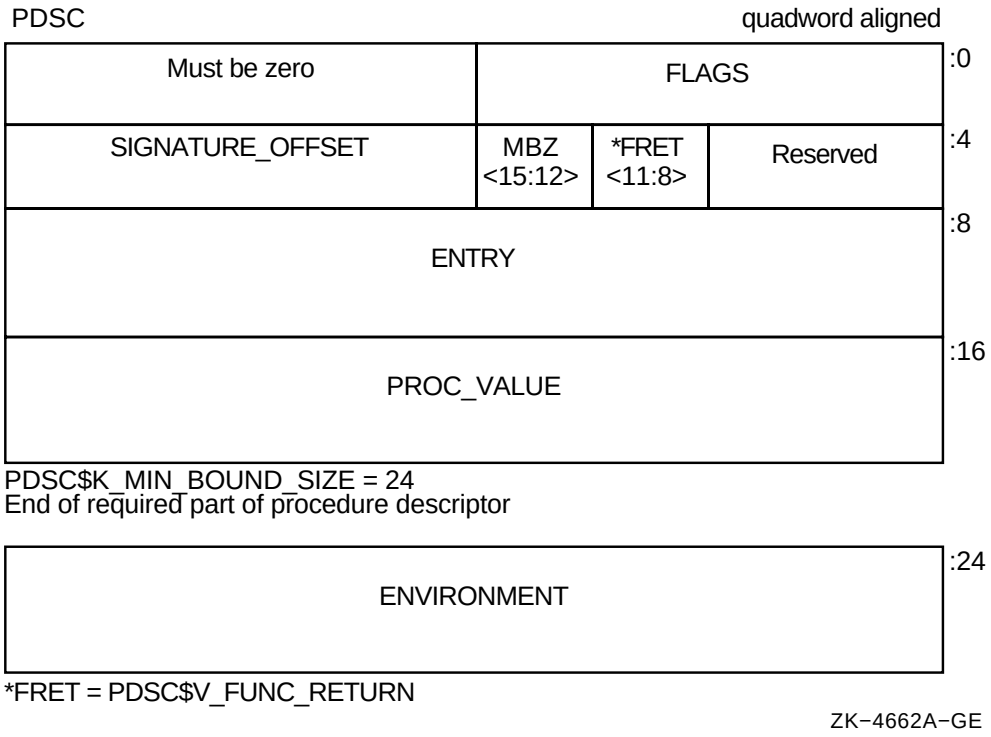


Table 3–13 Contents of the Bound Procedure Descriptor (PDSC)

Field Name	Contents
PDSC$W_FLAGS	Vector of flag bits <15:0> that must be a copy of the flag bits (except for KIND bits) contained in the quadword pointed to by PDSC$Q_PROC_VALUE.
	PDSC$V_KIND — A 4-bit field <3:0> that identifies the type of procedure descriptor. For a procedure with bound values, this field must specify a value of 0.
PDSC$V_FUNC_ RETURN	A 4-bit field <11:8> that describes which registers are used for the function value return (if there is one) and what format is used for those registers.
	PDSC$V_FUNC_RETURN in a bound procedure descriptor must be the same as the PDSC$V_FUNC_RETURN of the procedure descriptor for the procedure for which the environment is established.
	Table 3–7 lists and describes the possible encoding values of PDSC$V_FUNC_ RETURN.
Bits 12–15	Reserved and must be 0.

(continued on next page)

Table 3–13 (Cont.) Contents of the Bound Procedure Descriptor (PDSC)

Field Name	Contents
PDSC$W_SIGNATURE_OFFSET	A 16-bit signed byte offset from the start of the procedure descriptor. This offset designates the start of the procedure signature block (if any). In a bound procedure, a 0 in this field indicates the actual signature block must be sought in the procedure descriptor indicated by the PDSC$Q_PROC_VALUE field. A 1 in this field indicates a standard default signature. (An offset value of 1 is not a valid offset because both procedure descriptors and signature blocks must be quadword aligned. See Section 3.5 for details of the procedure signature block.) Note that a nonzero signature offset in a bound procedure value normally occurs only in the case of bound procedures used as part of the implementation of calls from native OpenVMS Alpha code to translated OpenVMS VAX images. In any case, if a nonzero offset is present, it takes precedence over signature information that might occur in any related procedure descriptor.
PDSC$Q_ENTRY	Address of the transfer code sequence.
PDSC$Q_PROC_VALUE	Value of the procedure to be called by the transfer code. The value can be either the address of a procedure descriptor for the procedure or possibly another bound procedure value.
PDSC$Q_ENVIRONMENT	An environment value to pass to the procedure. The choice of environment value is system implementation specific. For more information, see Section 3.7.4.1.

3.7.4.1 Bound Procedure Value

There are two distinct classes of procedures:

- Simple procedure
- Bound procedure

A **simple procedure** is a procedure that does not need direct access to the stack of its execution environment. A **bound procedure** is a procedure that does need direct access to the stack of its execution environment, typically to reference an up-level variable or to perform a nonlocal GOTO operation. Both a simple procedure and a bound procedure have an associated procedure descriptor, as described in previous sections.

When a bound procedure is called, the caller must pass some kind of pointer to the called code that allows it to reference its up-level environment. Typically, this pointer is the frame pointer for that environment, but many variations are possible. When the caller is executing its program within that outer environment, it can usually make such a call directly to the code for the nested procedure without recourse to any additional procedure descriptors. However, when a procedure value for the nested procedure must be passed outside of that environment to a call site that has no knowledge of the target procedure, a bound procedure descriptor is created so that the nested procedure can be called just like a simple procedure.

Bound procedure values, as defined by this standard, are designed for multilanguage use and utilize the properties of procedure descriptors to allow callers of procedures to use common code to call both bound and simple procedures.

The procedure value for a bound procedure is a pointer to a bound procedure descriptor that, like all other procedure descriptors, contains the address to which the calling procedure must transfer control at offset 8 (see Figure 3–13). This **transfer code** is responsible for setting up the dynamic environment needed by the target nested procedure and then completing the transfer of control to

the code for that procedure. The transfer code receives in R27 a pointer to its corresponding bound procedure descriptor and thus can fetch any required environment information from that descriptor. A bound procedure descriptor also contains a procedure value for the target procedure that is used to complete the transfer of control.

When the transfer code sequence addressed by PDSC$Q_ENTRY of a bound procedure descriptor is called (by a call sequence such as the one given in Section 3.7.3), the procedure value will be in R27, and the transfer code must finish setting up the environment for the target procedure. The preferred location for this transfer code is directly preceding the code for the target procedure. This saves a memory fetch and a branching instruction and optimizes instruction caches and paging.

The following is an example of such a transfer code sequence. It is an example of a target procedure Q that expects the environment value to be passed in R1 and a linkage pointer in R27.

```
Q_TRANSFER:
        LDQ     R1,24(R27)      ;Environment value to R1
        LDQ     R27,16(R27)     ;Procedure descriptor address to R27
Q_ENTRY::                       ;Normal procedure entry code starts here
```

After the transfer code has been executed and control is transferred to Q's entry address, R27 contains the address of Q's procedure descriptor, R26 (unmodified by transfer code) contains the return address, and R1 contains the environment value.

When a bound procedure value such as this is needed, the bound procedure descriptor is usually allocated on the parent procedure's stack.

3.7.5 Entry and Exit Code Sequences

To ensure that the stack can be interpreted at any point during thread execution, all procedures must adhere to certain conventions for entry and exit as defined in this section.

3.7.5.1 Entry Code Sequence

Since the value of FP defines the current procedure, all properties of the environment specified by a procedure's descriptor must be valid before the FP is modified to make that procedure current. In addition, none of the properties specified in the calling procedure's descriptor may be invalidated before the called procedure becomes current. So, until the FP has been modified to make the procedure current, all entry code must adhere to the following rules:

- All registers specified by this standard as saved across a standard call must contain their original (at entry) contents.
- No standard calls may be made.

Note

If an exception is raised or if an exception occurs in the entry code of a procedure, that procedure's exception handler (if any) will *not* be invoked since the procedure is not current yet. Therefore, if a procedure has an exception handler, compilers may not move code into the procedure prologue that might cause an exception that would be handled by that handler.

When a procedure is called, the code at the entry address must synchronize (as needed) any pending exceptions caused by instructions issued by the caller, must save the caller's context, and must make the called procedure current by modifying the value of FP as described in the following steps:

1. If PDSC$L_SIZE is not 0, set register SP = SP − PDSC$L_SIZE.
2. If PDSC$V_BASE_REG_IS_FP is 1, store the address of the procedure descriptor at 0(SP).

   If PDSC$V_KIND = PDSC$K_KIND_FP_REGISTER, copy the return address to the register specified by PDSC$B_SAVE_RA, if it is not already there, and copy the FP register to the register specified by PDSC$B_SAVE_FP.

   If PDSC$V_KIND = PDSC$K_KIND_FP_STACK, copy the return address to the quadword at the RSA$Q_SAVED_RETURN offset in the register save area denoted by PDSC$W_RSA_OFFSET, and store the registers specified by PDSC$L_IREG_MASK and PDSC$L_FREG_MASK in the register save area denoted by PDSC$W_RSA_OFFSET. (This step includes saving the value in FP.)

   Execute TRAPB if required (see Section 6.5.3.2 for details).
3. If PDSC$V_BASE_REG_IS_FP is 0, load register FP with the address of the procedure descriptor or the address of a quadword that contains the address of the procedure descriptor.

   If PDSC$V_BASE_REG_IS_FP is 1, copy register SP to register FP.

The ENTRY_LENGTH value in the procedure descriptor provides information that is redundant with the setting of a new frame pointer register value. That is, the value could be derived by starting at the entry address and scanning the instruction stream to find the one that updates FP. The ENTRY_LENGTH value included in the procedure descriptor supports the debugger or PCA facility so that such a scan is not required.

Entry Code Example for a Stack Frame Procedure

Example 3–2 is an entry code example for a stack frame. The example assumes that:

- This is a stack frame procedure
- Registers R2–4 and F2–3 are saved and restored
- PDSC$W_RSA_OFFSET = 16
- The procedure has a static exception handler that does not reraise arithmetic traps
- The procedure uses a variable amount of stack

If the code sequence in Example 3–2 is interrupted by an asynchronous software interrupt, SP will have a different value than it did at entry, but the calling procedure will still be current.

After an interrupt, it would not be possible to determine the original value of SP by the register frame conventions. If actions by an exception handler result in a nonlocal GOTO call to a location in the immediate caller, then it will not be possible to restore SP to the correct value in that caller. Therefore, any procedure that contains a label that can be the target of a nonlocal GOTO by immediately called procedures must be prepared to reset or otherwise manage the SP at that label.

Example 3–2 Entry Code for a Stack Frame Procedure

```
LDA     SP,-SIZE(SP)    ;Allocate space for new stack frame
STQ     R27,(SP)        ;Set up address of procedure descriptor
STQ     R26,16(SP)      ;Save return address
STQ     R2,24(SP)       ;Save first integer register
STQ     R3,32(SP)       ;Save next integer register
STQ     R4,40(SP)       ;Save next integer register
STQ     FP,48(SP)       ;Save caller's frame pointer
STT     F2,56(SP)       ;Save first floating-point register
STT     F3,64(SP)       ;Save last floating-point register
TRAPB                   ;Force any pending hardware exceptions to
                        ; be raised
MOV     SP,FP           ;Called procedure is now the current procedure
```

Entry Code Example for a Register Frame

Example 3–3 assumes that the called procedure has no static exception handler and utilizes no stack storage, PDSC$B_SAVE_RA specifies R26, PDSC$B_SAVE_FP specifies R22, and PDSC$V_BASE_REG_IS_FP is 0:

Example 3–3 Entry Code for a Register Frame Procedure

```
MOV     FP,R22          ;Save caller's FP.
MOV     R27,FP          ;Set FP to address of called procedure's
                        ; descriptor. Called procedure is now the
                        ; current procedure.
```

3.7.5.2 Exit Code Sequence

When a procedure returns, the exit code must restore the caller's context, synchronize any pending exceptions, and make the caller current by modifying the value of FP. The exit code sequence must perform the following steps:

1. If PDSC$V_BASE_REG_IS_FP is 1, then copy FP to SP.

   If PDSC$V_KIND = PDSC$K_KIND_FP_STACK, and this procedure saves or restores any registers other than FP and SP, reload those registers from the register save area as specified by PDSC$W_RSA_OFFSET.

   If PDSC$V_KIND = PDSC$K_KIND_FP_STACK, load a scratch register with the return address from the register save area as specified by PDSC$W_RSA_OFFSET. (If PDSC$V_KIND = PDSC$K_KIND_FP_REGISTER, the return address is already in scratch register PDSC$B_SAVE_RA.)

   Execute TRAPB if required (see Section 6.5.3.2 for details).

2. If PDSC$V_KIND = PDSC$K_KIND_FP_REGISTER, copy the register specified by PDSC$B_SAVE_FP to register FP.

3. If PDSC$V_KIND = PDSC$K_KIND_FP_STACK, reload FP from the saved FP in the register save area.

4. If a function value is not being returned using the stack (PDSC$V_STACK_RETURN_VALUE is 0), then restore SP to the value it had at procedure entry by adding the value that was stored in PDSC$L_SIZE to SP. (In some cases, the returning procedure will leave SP pointing to a lower stack address than it had on entry to the procedure, as specified in Section 3.8.7.)

5. Jump to the return address (which is in a scratch register).

The called routine does not adjust the stack to remove any arguments passed in memory. This responsibility falls to the calling routine that may choose to defer their removal because of optimizations or other considerations.

Exit Code Example for a Stack Frame

Example 3–4 shows the return code sequence for the stack frame.

Example 3–4 Exit Code Sequence for a Stack Frame

```
MOV     FP,SP           ;Chop the stack back
LDQ     R28,16(FP)      ;Get return address
LDQ     R2,24(FP)       ;Restore first integer register
LDQ     R3,32(FP)       ;Restore next integer register
LDQ     R4,40(FP)       ;Restore next integer register
LDT     F2,56(FP)       ;Restore first floating-point register
LDT     F3,64(FP)       ;Restore last floating-point register
TRAPB                   ;Force any pending hardware exceptions to
                        ; be raised
LDQ     FP,48(FP)       ;Restore caller's frame pointer
LDA     SP,SIZE(SP)     ;Restore SP (SIZE is compiled into PDSC$L_SIZE)
RET     R31,(R28)       ;Return to caller's code
```

Interruption of the code sequence in Example 3–4 by an asynchronous software interrupt can result in the calling procedure being the current procedure, but with SP not yet restored to its value in that procedure. The discussion of that situation in entry code sequences applies here as well.

Exit Code Example for a Register Frame

Example 3–5 contains the return code sequence for the register frame.

Example 3–5 Exit Code Sequence for a Register Frame

```
MOV     R22,FP          ;Restore caller's FP value
                        ; Caller is once again the current procedure.
RET     R31,(R26)       ;Return to caller's code
```

3.8 Data Passing

This section defines the OpenVMS Alpha calling standard conventions of passing data between procedures in a call chain. An argument item represents one unit of data being passed between procedures.

3.8.1 Argument-Passing Mechanisms

This OpenVMS Alpha calling standard defines three classes of argument items according to the mechanism used to pass the argument:

- Immediate value
- Reference
- Descriptor

Argument items are not self-defining; interpretation of each argument item depends on agreement between the calling and called procedures.

This standard does not dictate which passing mechanism must be used by a given language compiler. Language semantics and interoperability considerations might require different mechanisms in different situations.

Immediate value

An **immediate value** argument item contains the value of the data item. The argument item, or the value contained in it, is directly associated with the parameter.

Reference

A **reference** argument item contains the address of a data item such as a scalar, string, array, record, or procedure. This data item is associated with the parameter.

Descriptor

A **descriptor** argument item contains the address of a descriptor, which contains structural information about the argument's type (such as array bounds) and the address of a data item. This data item is associated with the parameter.

3.8.2 Argument List Structure

The argument list in an OpenVMS Alpha call is an ordered set of zero or more argument items, which together comprise a logically contiguous structure known as the argument item sequence. An argument item is specified using up to 64 bits.

A 64-bit argument item can be used to pass arguments by immediate value, by reference, and by descriptor. Any combination of these mechanisms in an argument list is permitted.

Although the argument items form a logically contiguous sequence, they are in practice mapped to integer and floating-point registers and to memory in a method that can produce a physically discontiguous argument list. Registers R16–21 and F16–21 are used to pass the first six items of the argument item sequence. Additional argument items must be passed in a memory argument list that must be located at 0(SP) at the time of the call.

Table 3–14 specifies the standard locations in which argument items can be passed.

Table 3–14 Argument Item Locations

Item	Integer Register	Floating-Point Register	Stack
1	R16	F16	
2	R17	F17	
3	R18	F18	
4	R19	F19	
5	R20	F20	
6	R21	F21	
7–*n*			0(SP) – (*n* – 7) * 8(SP)

The following list summarizes the general requirements that determine the location of any specific argument:

- All argument items are passed in the integer registers or on the stack, *except* for argument items that are floating-point data passed by immediate value.
- Floating-point data passed by immediate value is passed in the floating-point registers or on the stack.
- Only *one* location (across an item row in Table 3–14) can be used by any given argument item in a list. For example, if argument item 3 is an integer passed by value, and argument item 4 is a single-precision floating-point number passed by value, then argument item 3 is assigned to R18 and argument item 4 is assigned to F19.
- A single- or double-precision complex value is treated as two arguments for the purpose of argument-item sequence rules. In particular, the real part of a complex value might be passed as the sixth argument item in register F21, in which case the imaginary part is then passed as the seventh argument item in memory.

  An extended precision complex value is passed by reference using a single integer or stack argument item. (An extended precision complex value is not passed by immediate value because the component extended precision floating values are not passed by value. See also Section 3.8.5.1, Sending Mechanism.)

The argument list that includes both the in-memory portion and the portion passed in registers can be read from and written to by the called procedure. Therefore, the calling procedure must not make any assumptions about the validity of any part of the argument list after the completion of a call.

3.8.3 Argument Lists and High-Level Languages

High-level language functional notations for procedure call arguments are mapped into argument item sequences according to the following requirements:

- Arguments are mapped from left to right to increasing offsets in the argument item sequence. R16 or F16 is allocated to the first argument, and the last quadword of the memory argument list (if any) is allocated to the last argument.
- Each source language argument corresponds to one or more contiguous Alpha calling standard argument items.
- Each argument item consists of 64 bits.
- A null or omitted argument—for example, CALL SUB(A,,B)—is represented by an argument item containing the value 0.

  Arguments passed by immediate value cannot be omitted unless a default value is supplied by the language. (This is to enable called procedures to distinguish an omitted immediate argument from an immediate value argument with the value 0.)

  Trailing null or omitted arguments—for example, CALL SUB(A,,)—are passed by the same rules as for embedded null or omitted arguments.

3.8.4 Unused Bits in Passed Data

Whenever data is passed by value between two procedures in registers (for the first six input arguments and return values), or in memory (for arguments after the first six), the bits not used by the data are sign extended or zero extended as appropriate.

Table 3–15 lists and defines the various data-type requirements for size and their extensions to set or clear unused bits.

Table 3–15 Data Types and the Unused Bits in Passed Data

Data Type	Type Designator	Data Size (bytes)	Register Extension Type	Memory Extension Type
Byte logical	BU	1	Zero64	Zero64
Word logical	WU	2	Zero64	Zero64
Longword logical	LU	4	Sign64	Sign64
Quadword logical	QU	8	Data64	Data64
Byte integer	B	1	Sign64	Sign64
Word integer	W	2	Sign64	Sign64
Longword integer	L	4	Sign64	Sign64
Quadword integer	Q	8	Data64	Data64
F_floating	F	4	Hard	Data32
D_floating	D	8	Hard	Data64
G_floating	G	8	Hard	Data64
F_floating complex	FC	2 * 4	2*Hard	2*Data32
D_floating complex	DC	2 * 8	2*Hard	2*Data64
G_floating complex	GC	2 * 8	2*Hard	2*Data64
S_floating	FS	4	Hard	Data32
T_floating	FT	8	Hard	Data64
X_floating	FX	16	N/A	N/A
S_floating complex	FSC	2 * 4	2*Hard	2*Data32
T_floating complex	FTC	2 * 8	2*Hard	2*Data64
X_floating complex	FXC	2 * 16	N/A	N/A
Small structures of 8 bytes or less	N/A	$\leq$8	Nostd	Nostd
Small arrays of 8 bytes or less	N/A	$\leq$8	Nostd	Nostd
32-bit address	N/A	4	Sign64	Sign64
64-bit address	N/A	8	Data64	Data64

The following are the defined meanings for the extension type symbols used in Table 3–15:

Sign Extension Type	Defined Function
Sign64	Sign-extended to 64 bits.
Zero64	Zero-extended to 64 bits.
Data32	Data is 32 bits. The state of bits <63:32> is unpredictable.
2∗Data32	Two single-precision parts of the complex value are stored in memory as independent floating-point values (each handled as Data32).
Data64	Data is 64 bits.
2∗Data64	Two double-precision parts of the complex value are stored in memory as independent floating-point values (each handled as Data64).
Hard	Passed in the layout defined by the hardware SRM.
2∗Hard	Two floating-point parts of the complex value are stored in a pair of registers as independent floating-point values (each handled as Hard).
Nostd	State of all high-order bits not occupied by the data is unpredictable across a call or return.

Because of the varied rules for sign extension of data when passed as arguments, both calling and called routines must agree on the data type of each argument. No implicit data-type conversions can be assumed between the calling procedure and the called procedure.

3.8.5 Sending Data

This section defines the OpenVMS Alpha calling standard requirements for mechanisms to send data and the order of argument evaluation.

3.8.5.1 Sending Mechanism

As previously defined, the argument-passing mechanisms allowed are immediate value, reference, and descriptor. Requirements for using these mechanisms follow:

- **By immediate value.** An argument may be passed by immediate value only if the argument is one of the following:
  - One of the noncomplex scalar data types with a size known (at compile time) to be $\leq$ 64 bits
  - Either single or double precision complex
  - A record with a known size (at compile time)
  - A set, implemented as a bit vector, with a size known (at compile time) to be $\leq$ 64 bits

  No form of string or array data type may be passed by immediate value in a standard call.

  Unused high-order bits must be zero or sign extended, as appropriate depending on the date type, to fill all bits of each argument list item (as specified in Table 3–15).

  A single- or double- precision complex value is passed as two single or double precision floating-point values, respectively. Note that the argument count reflects that two argument positions are used rather than just one actual argument.

  A record value, which may be larger than 64 bits, is passed by immediate value as follows:

  - Allocate as many fully occupied argument item positions to the argument value as are needed to represent the argument.
  - The value of the unoccupied bits is undefined in a final, partially occupied argument item position, if any.
  - If an argument position is passed in one of the registers, it can only be passed in an integer register (never in a floating-point register).

  Other argument values that are larger than 64 bits can be passed by immediate value using nonstandard conventions, typically using a method similar to those for passing records. Thus, for example, a 26-byte string can be passed by value in four integer registers.

- **By reference.** Nonparametric arguments (arguments for which associated information such as string size and array bounds are not required) can be passed by reference in a standard call. This includes extended precision floating and extended precision complex values.
- **By descriptor.** Parametric arguments (arguments for which associated information such as string size and array bounds must be passed to the caller) are passed by a single descriptor in a standard call.

Note that extended floating values are not passed using the immediate value mechanism; rather, they are passed using the by reference mechanism. (However, when by value semantics is required, it may be necessary to make a copy of the actual parameter and pass a reference to that copy in order to avoid improper alias effects.)

Also note that when a record is passed by immediate value, the component types are not material to how the argument is aligned; the record will always be quadword aligned.

3.8.5.2 Order of Argument Evaluation

Since most high-level languages do not specify the order of evaluation (with respect to side effects) of arguments, those language processors can evaluate arguments in any convenient order. The choice of argument evaluation order and code generation strategy is constrained only by the definition of the particular language. Programs should not depend on the order of evaluation of arguments.

3.8.6 Receiving Data

When it cannot be determined at compile time whether a given in-register argument item is passed in a floating-point register or an integer register, the argument information register can be interpreted at run time to establish where the argument was passed. (See Section 3.7.1 for details.)

3.8.7 Returning Data

A standard function must return its function value by one of the following mechanisms:

- Immediate value
- Reference
- Descriptor

These mechanisms are the only standard means available for returning function values, and they support the important language-independent data types. Functions that return values by any mechanism other than those specified here are nonstandard, language-specific functions.

3.8.7.1 Function Value Return by Immediate Value

This standard defines the following two types of function returns by immediate value:

- Nonfloating function value return
- Floating function value return

Nonfloating Function Value Return by Immediate Value

A function value is returned by immediate value in register R0 *only* if the type of function value is one of the following:

- Nonfloating-point scalar data type with size known to be $\leq$ 64 bits
- Record with size known to be $\leq$ 64 bits
- Set, implemented as a bit vector, with size known to be $\leq$ 64 bits

No form of string or array can be returned by immediate value, and two separate 32-bit entities cannot be combined and returned in R0.

A function value of less than 64 bits returned in R0 must be zero extended or sign extended as appropriate, depending on the data type (see Table 3–15), to a full quadword.

Floating Function Value Return by Immediate Value

A function value is returned by immediate value in register F0 *only* if it is a noncomplex single- or double-precision floating-point value (F, D, G, S, or T).

A function value is returned by immediate value in registers F0 and F1 *only* if it is a complex single or double-precision floating-point value (complex F, D, G, S, or T).

Note that extended floating point and extended complex values are returned by reference as described next.

3.8.7.2 Function Value Return by Reference

A function value is returned by reference *only* if the function value satisfies both of the following criteria:

- Its size is known to both the calling procedure and the called procedure, but the value cannot be returned by immediate value. (Because the function value requires more than 64 bits, the data type is a string or an array type.)
- It can be returned in a contiguous region of storage.

The actual-argument list and the formal-argument list are shifted to the right by one argument item. The new, first argument item is reserved for the function value. This hidden first argument is included in the count and register usage information that is passed in the argument information register (see Section 3.7.1 for details).

The calling procedure must provide the required contiguous storage and pass the address of the storage as the first argument. This address *must* specify storage naturally aligned according to the data type of the function value.

The called function must write the function value to the storage described by the first argument.

3.8.7.3 Function Value Return by Descriptor

A function value is returned by descriptor *only* if the function value satisfies all of the following criteria:

- It cannot be returned by immediate value. (Because the function value requires more than 64 bits, the data type is a string or an array type, and so on.)
- Its size is not known to either the calling procedure or the called procedure.
- It can be returned in a contiguous region of storage.

Noncontiguous function values are language specific and cannot be returned as a standard-conforming return value.

Records, noncontiguous arrays, and arrays with more than one dimension cannot be returned by descriptor in a standard call.

Both 32-bit and 64-bit descriptor forms can be used for function values returned by descriptor. See Chapter 5 for details of the descriptor forms.

The use of descriptors for function value return divides into three major cases with return values involving:

- Dynamic text—Heap-managed strings of arbitrary and dynamically changeable length
- Return objects created by the calling routine—Function values that are to be returned in an object allocated by and having attributes (bounds, lengths, and so on) specified by the calling routine
- Return objects created by the called routine—Function values that are returned in an object allocated by and having attributes (bounds, lengths, and so on) specified by the called routine

For correct results to be obtained from this type of function return, the calling and called routines must agree by prior arrangement which of these three major cases applies, and whether 64-bit descriptor forms may be used.

The following paragraphs describe the specialized requirements for each major case:

Dynamic Text

For dynamic text return by descriptor, the calling routine passes a valid (completely initialized) dynamic string descriptor (DSC$B_CLASS = DSC$K_ CLASS_D). The called routine must assign a value to the variable represented by this descriptor using the same rules that apply to a dynamic text descriptor used as an ordinary parameter.

Return Object Created by Calling Routine

For a return object created by the calling routine, the calling routine passes a descriptor in which all fields are completely loaded.

The called routine must supply a return value that satisfies that description. In particular, the called routine must truncate or pad the returned value to satisfy the requirements of the descriptor according to the semantics of the language in which the called routine is written.

The calling and called routines must agree by prior arrangement on the DSC$B_ CLASS and DSC$B_DTYPE of descriptor to be used.

Return Object Created by Called Routine

For a return object created by the called routine, the calling and called routines must agree by prior arrangement on the DSC$B_CLASS and DSC$B_DTYPE of descriptor to be used. The calling routine passes a descriptor in which:

- DSC$A_POINTER field is set to 0.
- DSC$B_CLASS field is loaded.
- DSC$B_DTYPE field is loaded.
- DSC$B_DIMCT field is loaded and the DSC$B_AFLAGS field is set to 0 if the descriptor is an array descriptor.
- All other fields are unpredictable.

If the passed descriptor is an array descriptor, it must contain space for bounds information to be returned even though the DSC$B_AFLAGS field is set to 0.

The called routine must return the function value using stack return conventions and load the DSC$A_POINTER field to point to the returned data. Other descriptor information, such as origin, bounds (if supplied), and DSC$B_AFLAGS fields must be filled in appropriately to correspond to the returned data.

An important implication of a call that uses this kind of value return is that the stack pointer normally is not restored to its value prior to the call as part of the return from the called procedure. The returned value typically (but not necessarily) is left by the called routine somewhere on the stack. For that reason, this mechanism is sometimes known as the **stack return** mechanism.

However, this type of return does not imply that the actual storage used by the called routine to hold the returned value must be at the address pointed to by the stack pointer; it need not even be on the stack. It could be in some read-only, static memory. (This latter case might arise when the returned value is constant or is obtained from some constant structure.) For this reason, the calling routine must not assume that the data described by the return descriptor is writable.

3.9 Static Data

This section describes the standard static data requirements that define the Alpha alignment of data structures, record formats, and record layout. These conventions help to ensure proper data compatibility with all OpenVMS Alpha and VAX languages.

3.9.1 Alignment

In the Alpha environment, memory references to data that is not naturally aligned can result in alignment faults, which can severely degrade the performance of all procedures that reference the unaligned data.

To avoid such performance degradation, all data values on Alpha systems should be naturally aligned. Table 3–16 contains information on data alignment.

Table 3–16 Data Alignment Addresses

Data Type	Alignment Starting Position
8-bit character string	Byte boundary
16-bit integer	Address that is a multiple of 2 (word alignment)
32-bit integer	Address that is a multiple of 4 (longword alignment)
64-bit integer	Address that is a multiple of 8 (quadword alignment)
F_floating F_floating complex	Address that is a multiple of 4 (longword)
D_floating D_floating complex	Address that is a multiple of 8 (quadword)
G_floating G_floating complex	Address that is a multiple of 8 (quadword)
S_floating S_floating complex	Address that is a multiple of 4 (longword alignment)
T_floating T_floating complex	Address that is a multiple of 8 (quadword)
X_floating X_floating complex	Address that is a multiple of 16 (octaword)

For aggregates such as strings, arrays, and records, the data type to be considered for purposes of alignment is *not* the aggregate itself, but rather the elements of which the aggregate is composed. The alignment requirement of an aggregate is that all elements of the aggregate be naturally aligned. For example, varying 8-bit character strings must start at addresses that are a multiple of at least 2 (word alignment) because of the 16-bit count at the beginning of the string; 32-bit integer arrays start at a longword boundary, irrespective of the extent of the array.

The rules for passing a record in an argument that is passed by immediate value (see Section 3.8.5.1) always provide quadword alignment of the record value independent of the normal alignment requirement of the record. If deemed appropriate by an implementation, normal alignment can be established within the called procedure by making a copy of the record argument at a suitably aligned location.

3.9.2 Record Layout Conventions

The OpenVMS Alpha calling standard rules for record layout are designed to provide good run-time performance on all implementations of the Alpha architecture and to provide the required level of compatibility with conventional VAX operating environments.

Therefore, this standard defines two record layout conventions:

- Those optimized for optimal access characteristics (referred to as **aligned** record layouts)
- Those compatible with conventions that are traditionally used by VAX languages (referred to as **VAX compatible** record layouts)

Note

Although compiler implementers must make appropriate business decisions, Compaq strongly recommends that all Alpha high-level language compilers should support both record layouts.

Only these two record layouts may be used across standard interfaces or between languages. Languages can support other language-specific record layout conventions, but such layouts are nonstandard.

The aligned record layout conventions should be used unless interchange is required with conventional VAX applications that use the VAX VMS or OpenVMS VAX compatible record layouts.

3.9.2.1 Aligned Record Layout

The aligned record layout conventions ensure that:

- All components of a record or subrecord are naturally aligned.
- Layout and alignment of record elements and subrecords are independent of any record or subrecord in which they are embedded.
- Layout and alignment of a subrecord is the same as if it were a top-level record.
- Declaration in high-level languages of standard records for interlanguage use is straightforward and obvious, and meets the requirements for source-level compatibility between Alpha and VAX languages.

The aligned record layout is defined by the following conventions:

- The components of a record must be laid out in memory corresponding to the lexical order of their appearance in the high-level language declaration of the record.
- The first bit of a record or subrecord must be directly addressable (byte aligned).
- Records and subrecords must be aligned according to the largest natural alignment requirements of the contained elements and subrecords.
- Bit fields (packed subranges of integers) are characterized by an underlying integer type that is a byte, word, longword, or quadword in size together with an allocation size in bits. A bit field is allocated at the next available bit boundary, provided that the resulting allocation does not cross an alignment boundary of the underlying type. Otherwise, the field is allocated at the next byte boundary that is aligned as required for the underlying type. (In the later case, the space skipped over is left permanently not allocated.) In addition, if necessary, the alignment of the record as a whole is increased to that of the underlying integer type.
- Unaligned bit strings, unaligned bit arrays, and elements of unaligned bit arrays must start at the next available bit in the record. No fill is ever supplied preceding an unaligned bit string, unaligned bit array, or unaligned bit array element.
- All other components of a record must start at the next available naturally aligned address for the data type.

- The length of a record must be a multiple of its alignment. (This includes the case when a record is a component of another record.)
- Strings and arrays must be aligned according to the natural alignment requirements of the data type of which the string or array is composed.
- The length of an array element is a multiple of its alignment, even if this leaves unused space at its end. The length of the whole array is the sum of the lengths of its elements.

3.9.2.2 OpenVMS VAX Compatible Record Layout

The OpenVMS VAX compatible record layout is defined by the following conventions:

- The components of a record must be laid out in memory corresponding to the lexical order of their appearance in the high-level language declaration of the record.
- Unaligned bit strings, unaligned bit arrays, and elements of unaligned bit arrays must start at the next available bit in the record. No fill is ever supplied preceding an unaligned bit string, unaligned bit array, or unaligned bit array element.
- All other components of a record must start at the next available byte in the record. Any unused bits following the last-used bit in the last-used byte of each component must be filled out to the next byte boundary so that any following data starts on a byte boundary.
- Subrecords must be aligned according to the largest alignment of the contained elements and subrecords. A subrecord always starts at the next available byte unless it consists entirely of unaligned bit data and it immediately follows an unaligned bit string, unaligned bit array, or a subrecord consisting entirely of unaligned bit data.
- Records must be aligned on byte boundaries.

3.10 Multithreaded Execution Environments

This section defines the conventions to support the execution of multiple threads in a multilanguage Alpha environment. Specifically defined is how compiled code must perform stack limit checking. While this standard is compatible with a multithreaded execution environment, the detailed mechanisms, data structures, and procedures that support this capability are not specified in this manual.

For a multithread environment, the following characteristics are assumed:

- There can be one or more threads executing within a single process.
- The state of a thread is represented in a **thread environment block (TEB)**.
- The TEB of a thread contains information that determines a stack limit below which the stack pointer must not be decremented by the executing code (except for code that implements the multithread mechanism itself).
- Exception handling is fully reentrant and multithreaded.
- The correct way to terminate a thread is by returning from the initial procedure in which the thread begins execution, or by a call to SYS$GOTO_UNWIND, specifying a null target environment or some other procedure that includes this effect. Note that correct thread termination involves unwind processing for all of the active frames of the thread.

3.10.1 Stack Limit Checking

A program that is otherwise correct can fail because of stack overflow. Stack overflow occurs when extension of the stack (by decrementing the stack pointer, SP) allocates addresses not currently reserved for the current thread's stack.

Detection of a stack overflow situation is necessary because a thread, attempting to write into stack storage, could modify data allocated in that memory for some other purpose. This would most likely produce unpredictable and undesirable results or irreproducible application failures.

The requirements for procedures that can execute in a multithread environment include checking for stack overflow. This section defines the conventions for stack limit checking in a multithreaded program environment.

In the following sections, the term **new stack region** refers to the region of the stack from the old value of SP − 1 to the new value of the SP.

3.10.1.1 Stack Guard Region

In a multithread environment, the memory beyond the limit of each thread's stack is protected by contiguous **guard pages**, which form the stack's **guard region**.

3.10.1.2 Stack Reserve Region

In some cases, it is desirable to maintain a stack **reserve region**, which is a minimum-sized region that is immediately above a thread's guard region. A reserve region may be desirable to ensure that exceptions or asynchronous system traps (ASTs) have stack space to execute on a thread's stack, or to ensure that the exception dispatcher and any exception handler that it might call have stack space to execute after detection of an invalid attempt to extend the stack.

This standard does not require a reserve region.

3.10.1.3 Methods for Stack Limit Checking

Since accessible memory may be available at addresses lower than those occupied by the guard region, compilers must generate code that never extends the stack past the guard pages into accessible memory that is not allocated to the thread's stack.

A general strategy is to access each page of memory down to and possibly including the page corresponding to the intended new value for the SP. If the stack is to be extended by an amount larger than the size of a memory page, then a series of accesses is required that works from higher to lower addressed pages. If any access results in a memory access violation, then the code has made an invalid attempt to extend the stack of the current thread.

Note

An access can be performed by using either a load or a store operation; however, be sure to use an instruction that is guaranteed to make an access to memory. For example, do not use an `LDQ R31,*` instruction, because the Alpha architecture does not allow any memory access, even a read access, whose result is discarded because of the R31 destination.

This standard defines two methods for stack limit checking: implicit and explicit.

Implicit Stack Limit Checking

The following are two mutually exclusive strategies for implicit stack limit checking:

- If the lowest addressed byte of the new stack region is guaranteed to be accessed prior to any further stack extension, then the stack can be extended by an increment that is equal in size to the guard region (without any further accesses).
- If some byte (not necessarily the lowest) of the new stack region is guaranteed to be accessed prior to any further stack extension, then the stack can be extended by an increment that is equal in size to one-half the guard region (without any further accesses).

The stack frame format (see Section 3.4.3) and entry code rules (see Section 3.7.5) generally do not ensure access to the lowest address of a new stack region without introducing an extra access solely for that purpose. Consequently, this standard uses the second strategy. While the amount of implicit stack extension that can be achieved is smaller, the check is achieved at no additional cost.

This standard requires that the minimum guard region size is 8192 bytes, the size of the smallest memory protection granularity allowed by the Alpha architecture.

Therefore, if the stack is being extended by an amount less than or equal to 4096 and a reserve region is not required, then explicit stack limit checking is not required.

However, because asynchronous interrupts and calls to other procedures may also cause stack extension without explicit stack limit checking, stack extension with implicit limit checking must adhere to a strict set of conventions as follows:

1. Explicit stack limit checking must be performed unless the amount by which the SP is decremented is known to be less than or equal to 4096 and a reserve region is not required.
2. Some byte in the new stack region must be accessed before the SP can be decremented for a subsequent stack extension.

   This access can be performed either before or after the SP is decremented for this stack extension, but it must be done before the SP can be decremented again.
3. No standard procedure call can be made before some byte in the new stack region is accessed.
4. The system exception dispatcher ensures that the lowest addressed byte in the new stack region is accessed if any kind of asynchronous interrupt occurs after the SP is decremented, but before the access in the new stack region occurs.

These conventions ensure that the stack pointer is not decremented so that it points to accessible storage beyond the stack limit without this error being detected (either by the guard region being accessed by the thread or by an explicit stack limit check failure).

As a matter of practice, the system can provide multiple guard pages in the guard region. When a stack overflow is detected as a result of access to the guard region, one or more guard pages can be unprotected for use by the exception-handling facility, and one or more guard pages can remain protected to provide implicit stack limit checking during exception processing. However, the size of

the guard region and the number of guard pages is system defined and is not defined by this standard.

Explicit Stack Limit Checking

If the stack is being extended by an amount of unknown size or by a known size greater than the maximum implicit check size (4096), then a code sequence that follows the rules for implicit stack limit checking can be executed in a loop to access the new stack region incrementally in segments lesser than or equal to the minimum page size (8192 bytes). At least one access must occur in each such segment. The first access must occur between SP and SP – 4096 because, in the absence of more specific information, the previous guaranteed access relative to the current stack pointer may be as much as 4096 bytes greater than the current stack pointer address. The last access must be within 4096 bytes of the intended new value of the stack pointer. These accesses must occur in order, starting with the highest addressed segment and working toward the lowest addressed segment.

Note

A simple algorithm that is consistent with this requirement (but achieves up to twice the minimum number of accesses) is to perform a sequence of accesses in a loop starting with the previous value of SP, decrementing by the minimum no-check extension size (4096) to, but not including, the first value that is less than the new value for the stack pointer.

The stack must *not* be extended incrementally in procedure prologues. A procedure prologue that needs to extend the stack by an amount of unknown size or known size greater than the minimum implicit check size must test new stack segments as just described in a loop that does not modify SP, and then update the stack with one instruction that copies the new stack pointer value into the SP.

Note

An explicit stack limit check can be performed either by inline code that is part of a prologue or by a run-time support routine that is tailored to be called from a procedure prologue.

Stack Reserve Region Checking

The size of the reserve region must be included in the increment size used for stack limit checks, after which it is not included in the amount by which the stack is actually extended. (Depending on the size of the reserve region, this may partially or even completely eliminate the ability to use implicit stack limit checking.)

3.10.1.4 Stack Overflow Handling

If a stack overflow is detected, one of the following results:

- Exception SS$_ACCVIO may be raised.
- The system may transparently extend the thread's stack, reset the TEB stack limit value appropriately, and continue execution of the thread.

Note that if a transparent stack extension is performed, a stack overflow that occurs in a called procedure might cause the stack to be extended. Therefore, the TEB stack limit value must be considered volatile and potentially modified by external procedure calls and by handling of exceptions.

4

OpenVMS Argument Data Types

This chapter defines the argument-passing data types that are used to call a procedure for both VAX and Alpha environments. All features defined here apply to both OpenVMS VAX and OpenVMS Alpha systems unless otherwise noted.

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

- Atomic
- String
- Miscellaneous

When existing data types fail to satisfy the semantics of a language, new data types, including certain language-specific ones, are added to this standard. These data types can generally be passed by immediate value (if 32 bits or less), by reference, or by descriptor.

Each data type code presented in this chapter indicates a unique data format. Use these encodings whenever you need to identify data types to achieve greater commonality across user software.

The encoding given in Sections 4.1 and 4.2 can help you to identify data types, such as in a descriptor. However, in addition to their use in descriptors, these data type codes are also useful for identifying VAX and Alpha data types in areas outside the scope of the calling standard. Therefore, each data-type code indicates a unique data format independent of its use in descriptors.

Some data types are composed of a recordlike 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 in this standard represent signed quantities. The unsigned quantities do not allocate space for the sign; all bit or character positions are used for significant data.

4.1 Atomic Data Types

Table 4–1 shows how atomic data types are defined and encoded for VAX and Alpha environments.

Table 4–1 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$K_DTYPE_B	6	Byte integer (signed) 8-bit signed two's complement integer.
DSC$K_DTYPE_W	7	Word integer (signed) 16-bit signed two's complement integer.
DSC$K_DTYPE_L	8	Longword integer (signed) 32-bit signed two's complement integer.
DSC$K_DTYPE_Q	9	Quadword integer (signed) 64-bit signed two's complement integer.
DSC$K_DTYPE_O	26	Octaword integer (signed) 128-bit signed two's complement integer.
DSC$K_DTYPE_F	10	F_floating 32-bit F_floating quantity representing a single-precision number.
DSC$K_DTYPE_D[1]	11	D_floating 64-bit D_floating quantity representing a double-precision number.
DSC$K_DTYPE_G	27	G_floating 64-bit G_floating quantity representing a double-precision number.
†DSC$K_DTYPE_H[2]	28	H_floating 128-bit H_floating quantity representing a quadruple-precision number.

[1]While the calling standard supports the manipulation of D_floating and D_floating complex data, compiled code support will invoke conversion from D_floating to G_floating as needed for Alpha arithmetic operations, and conversion of G_floating intermediate results back to D_floating when needed for stores to memory or parameter passing. This allows D_floating data to be used in Alpha arithmetic operations without required source changes but with results limited to G_floating precision.

[2]H_floating data is not supported for general use on OpenVMS Alpha systems. However, conversion routines are supplied to allow users to convert existing H_floating data to other storage representations.

†VAX specific.

(continued on next page)

Table 4–1 (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[2]	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$K_DTYPE_FS	52	S_floating 32-bit IEEE S_floating quantity representing a single-precision number.
‡DSC$K_DTYPE_FT	53	T_floating 64-bit IEEE T_floating quantity representing a double-precision number.
‡DSC$K_DTYPE_FSC	54	S_floating complex Ordered pair of S_floating quantities representing a single-precision complex number. The lower addressed quantity is the real part; the higher addressed quantity is the imaginary part.
‡DSC$K_DTYPE_FTC	55	T_floating complex Ordered pair of T_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$K_DTYPE_FX	57	X_floating 128-bit IEEE X_floating quantity representing an extended-precision number.
‡DSC$K_DTYPE_FXC	58	X_floating complex Ordered pair of X_floating quantities representing an extended-precision complex number. The lower addressed quantity is the real part; the higher addressed quantity is the imaginary part.

[2]H_floating data is not supported for general use on OpenVMS Alpha systems. However, conversion routines are supplied to allow users to convert existing H_floating data to other storage representations.

†VAX specific.

‡Alpha specific.

4.2 String Data Types

String data types are ordinarily described by a string descriptor. Table 4–2 shows how the string data types are defined and encoded for OpenVMS VAX and OpenVMS Alpha environments.

Table 4–2 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).
DSC$K_DTYPE_VT	37	Varying character string A 16-bit unsigned count of the current number of 8-bit characters in the following string, followed by a string of 0 to $2^{16} - 1$ 8-bit characters (see Section 4.5 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 4.5, 5.8, and 5.9 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 0 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 5.10 and 5.11).

4.3 Miscellaneous Data Types

Table 4–3 shows how miscellaneous data types are defined and encoded for the OpenVMS VAX and OpenVMS Alpha environments.

Table 4–3 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 (for VAX environment only) A two-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$K_DTYPE_BLV	33	Bound label value A two-longword entity in which the first longword contains the address of an instruction and the second longword is the language-specific environment value. The environment value is determined in a language-specific manner when the original bound label value is generated.
DSC$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 OpenVMS 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 0 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 OpenVMS date format for positive values only.

†VAX specific.

4.4 Reserved Data-Type Codes

All codes from 0 through 191 not otherwise defined in this standard are reserved to Compaq. Codes 192 through 255 are reserved to Compaq's Computer Special Systems Group and for customers for their own use.

Table 4–4 lists the data types and codes that are obsolete or reserved to Compaq.

Table 4–4 Reserved Data Types

Symbol	Code	Purpose
DSC$K_DTYPE_CIT	31	Reserved to COBOL (intermediate temporary)
DSC$K_DTYPE_CIT2	64	Reserved to COBOL (intermediate temporary alternative 2)
DSC$K_DTYPE_TF	40	Reserved to DEBUG (Boolean true/false)
DSC$K_DTYPE_SV	41	Reserved to DEBUG (signed bit-field, aligned)
DSC$K_DTYPE_SVU	42	Reserved to DEBUG (signed bit-field, unaligned)
DSC$K_DTYPE_FIXED	43	Reserved to DEBUG (fixed binary—fixed point in Ada and fixed binary in PL/I)
DSC$K_DTYPE_TASK	44	Reserved to DEBUG (task type in Ada)
DSC$K_DTYPE_AC	45	Reserved to DEBUG (ASCIC text)
DSC$K_DTYPE_AZ	46	Reserved to DEBUG (ASCIZ text)
DSC$K_DTYPE_M68_S	47	Reserved to DEBUG (Motorola 68881 single precision, 32-bit)[1]
DSC$K_DTYPE_M68_D	48	Reserved to DEBUG (Motorola 68881 double precision, 64-bit)[1]
DSC$K_DTYPE_M68_X	49	Reserved to DEBUG (Motorola 68881 extended precision, 96-bit)[2]
DSC$K_DTYPE_1750_S	50	Reserved to DEBUG (1750 single precision, 32-bit)
DSC$K_DTYPE_1750_X	51	Reserved to DEBUG (1750 extended precision, 48-bit)
DSC$K_DTYPE_WC	56	Reserved to DEBUG (`setlocale` dependent C string)
No symbol defined	36	Obsolete
DSC$K_DTYPE_T2	38	Obsolete
DSC$K_DTYPE_VT2	39	Obsolete

[1]Differs from Alpha IEEE floating because of byte ordering.

[2]Differs from Alpha IEEE floating because of byte ordering and size.

4.4.1 Facility-Specific Data-Type Codes

Data-type codes 160 through 191 are reserved to Compaq 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 with the OpenVMS Debugger.

As shown in Table 4–4, data-type codes 31 and 64 are reserved for the COBOL facility. Codes 40 through 51 and 56 are reserved for the OpenVMS Debugger facility.

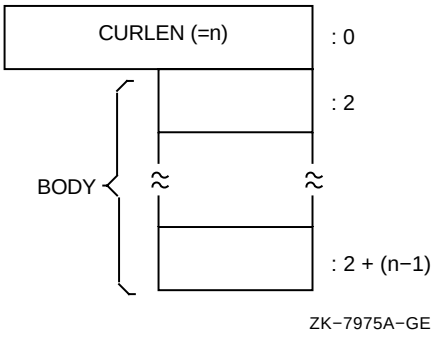
4.5 Varying Character String Data Type (DSC$K_DTYPE_VT)

The varying character string data type (DSC$K_DTYPE_VT) consists of the following two fixed-length areas allocated contiguously with no padding in between (see Figure 4–1):

CURLEN — An unsigned word specifying the current length in bytes of the immediately following string.

BODY — A fixed-length area containing the string that can vary from 0 to a maximum length defined for each instance of string. The range of this maximum length is 0 to $2^{16} - 1$.

Figure 4–1 Varying Character String Data Type (DSC$K_DTYPE_VT)—General Format

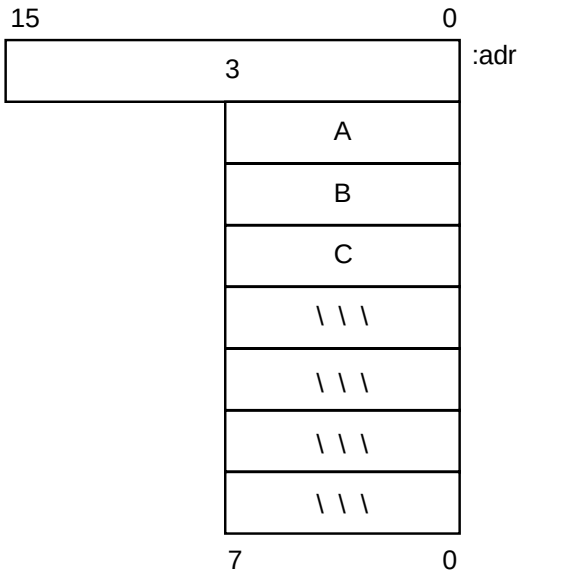


When passed by reference or by descriptor, the address of the varying character string (VT) data type is always the address of the CURLEN field, not the BODY field.

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. To represent the string ABC, CURLEN will have a value of 3 and the last four bytes will be undefined, as shown in Figure 4–2.

Figure 4–2 Varying Character String Data Type (DSC$K_DTYPE_VT) Format



ZK–1889–GE

5
OpenVMS Argument Descriptors

This chapter describes the argument descriptors used in calling a procedure for both VAX and Alpha environments.

A uniform descriptor mechanism is defined for use by all VAX and Alpha procedures that conform to the OpenVMS calling standard. Descriptors are self-describing and the mechanism is extensible. When existing descriptors fail 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 ignores the contents of some of the fields. Therefore, a descriptor conforms to this calling standard if all fields are filled in by the calling program, even if the called program does not need the field.

Note

Unless stated otherwise, all fields in descriptors represented as 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 Chapter 4), the processor is not required to use a standard descriptor to pass an array of such a data type. However, if a language processor passes an array of such a data type using a standard descriptor, the language processor fills in the DSC$B_DTYPE field with the value 0, 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 0 and for additional data types.

Table 5–1 identifies the classes of argument descriptors for use in the standard VAX and Alpha environments. Each class has two synonymous names—one for 32-bit environments (DSC$) and one for 64-bit environments (DSC64$). Descriptions and formats of each of these descriptors follow.

Table 5–1 Argument Descriptor Classes for OpenVMS Alpha and OpenVMS VAX

Descriptor	Code	Class
DSC$K_CLASS_S DSC64$K_CLASS_S	1	Fixed-length scalar/string
DSC$K_CLASS_D DSC64$K_CLASS_D	2	Dynamic string
DSC$K_CLASS_A DSC64$K_CLASS_A	4	Contiguous array
DSC$K_CLASS_P[1] DSC64$K_CLASS_P[1]	5	Procedure argument descriptor
DSC$K_CLASS_SD DSC64$K_CLASS_SD	9	Decimal (scalar) string
DSC$K_CLASS_NCA DSC64$K_CLASS_NCA	10	Noncontiguous array
DSC$K_CLASS_VS DSC64$K_CLASS_VS	11	Varying string
DSC$K_CLASS_VSA DSC64$K_CLASS_VSA	12	Varying string array
DSC$K_CLASS_UBS DSC64$K_CLASS_UBS	13	Unaligned bit string
DSC$K_CLASS_UBA DSC64$K_CLASS_UBA	14	Unaligned bit array
DSC$K_CLASS_SB DSC64$K_CLASS_SB	15	String with bounds
DSC$K_CLASS_UBSB DSC64$K_CLASS_UBSB	16	Unaligned bit string with bounds

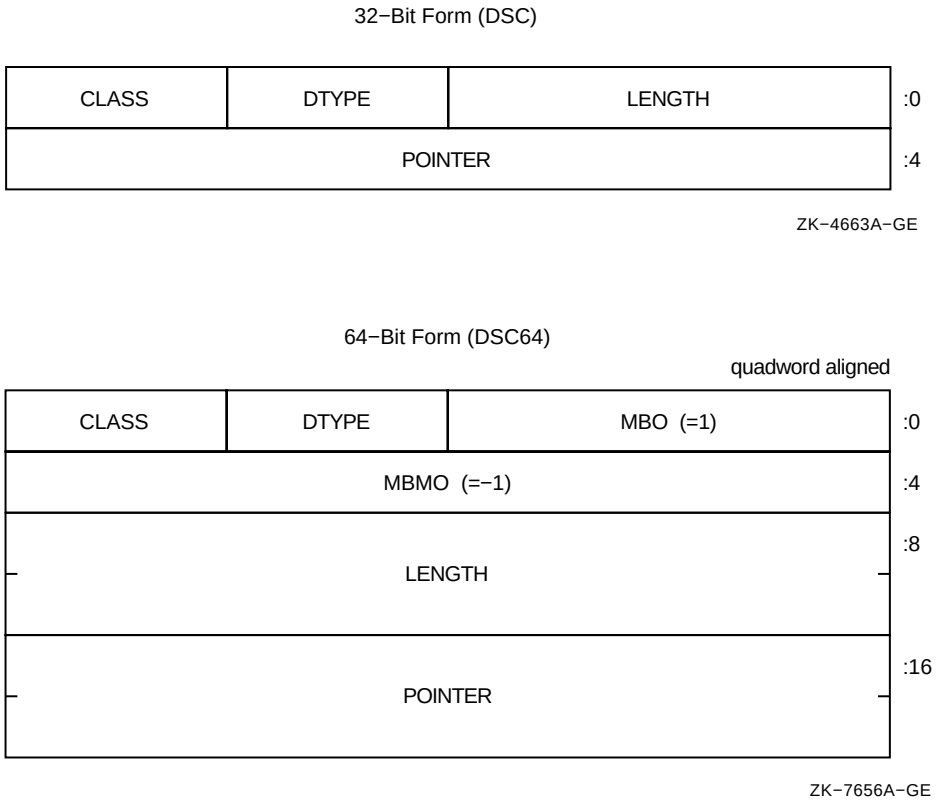
[1]The pointer field usage for this descriptor differs from VAX usage (see Section 5.5).

5.1 Descriptor Prototype

Figure 5–1 shows the descriptor prototype format. There are two forms: one for use with 32-bit addresses and one for use with 64-bit addresses. The two forms are compatible in that the forms can be distinguished dynamically at run time and, except for the size and consequential placement of fields, 32-bit and 64-bit descriptors are identical in content and interpretation.

The 32-bit descriptors are used on both OpenVMS VAX and OpenVMS Alpha systems. When used on OpenVMS Alpha systems, 32-bit descriptors provide full compatibility with their use on OpenVMS VAX. The 64-bit descriptors are used only on OpenVMS Alpha systems—they have no counterparts and are not recognized on OpenVMS VAX systems.

Figure 5–1 Descriptor Prototype Format



The 32-bit descriptors on OpenVMS Alpha systems have no required alignment for compatibility with OpenVMS VAX systems; however, longword alignment generally promotes performance. The 64-bit descriptors on OpenVMS Alpha systems must be quadword aligned.

Table 5–2 describes the fields of the descriptor. In this table and the similar tables for descriptors in later sections, note that most fields have two symbols and one description. The symbol that begins with the prefix DSC$ is used with 32-bit descriptors, while the symbol that begins with the prefix DSC64$ is used with 64-bit descriptors.

In this chapter, it is generally the practice to use only the main part of a field name, without either of the prefixes used in actual code. For example, the length field is referred to using LENGTH rather than mentioning both DSC$W_LENGTH and DSC64$Q_LENGTH. The DSC$ and DSC64$ prefixes are used only when referring to a particular form of descriptor.

The CLASS and DTYPE fields occupy the same offsets in both 32-bit and 64-bit descriptors. Thus, the symbols DSC$B_CLASS and DSC64$B_CLASS have the same definition, as do DSC$B_DTYPE and DSC64$B_DTYPE. Furthermore, these fields are permitted to contain the same values with the same meanings in both 32-bit and 64-bit forms.

The DSC$W_LENGTH and DSC$A_POINTER fields in the 32-bit descriptors correspond in placement to the DSC64$W_MBO (must be 1) and DSC64$L_ MBMO (must be -1) fields in the 64-bit descriptors. The values of these fields are used to distinguish whether a given descriptor has the 32-bit or 64-bit form as described later in this section.

When the CLASS field is 0, no more information can be assumed than is shown in Table 5–2.

Table 5–2 Contents of the Prototype Descriptor

Symbol	Description
DSC$W_LENGTH DSC64$Q_LENGTH	Defines the data item length specific to the descriptor class.
DSC64$W_MBO	In a 64-bit descriptor, this field must contain the value 1. This field overlays the DSC$W_LENGTH field of a 32-bit descriptor and the value 1 is necessary to correctly distinguish between the two forms (see below).
DSC$B_DTYPE DSC64$B_DTYPE	A data-type code. Data-type codes are listed in Sections 4.1 and 4.2.
DSC$B_CLASS DSC64$B_CLASS	A 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 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$A_POINTER DSC64$PQ_POINTER	The address of the first byte of the data element described.
DSC64$L_MBMO	In a 64-bit descriptor, this field must contain the value -1 (all 1 bits). Note that this field overlays the DSC$A_POINTER field of a 32-bit descriptor and the value -1 is necessary to correctly distinguish between the two forms (see below).

As previously mentioned, the MBO field (a word at offset 0) and the MBMO field (a longword at offset 4) are used to distinguish between a 32-bit and 64-bit descriptor. A called routine that is designed to handle both kinds of descriptors must do both of the following:

- Confirm that the MBO field contains 1
- Confirm that the MBMO field contains -1

before concluding that it has a 64-bit form descriptor.

Note

It may seem sufficient to test just the MBMO field. However, that allows a 32-bit descriptor with a length of 0 and an undefined pointer to be inadvertently treated as a 64-bit descriptor.

If the MBMO field contains -1, then 0 and 1 are the only values of the MBO field that have defined interpretations.

5.2 Fixed-Length Descriptor (CLASS_S)

A single descriptor class is used for scalar data and fixed-length strings. Any OpenVMS data type, except data type 34 (unaligned bit string), can be used with this descriptor. Figure 5–2 shows the format of a fixed-length descriptor. Table 5–3 describes the fields of the descriptor.

Figure 5–2 Fixed-Length Descriptor Format

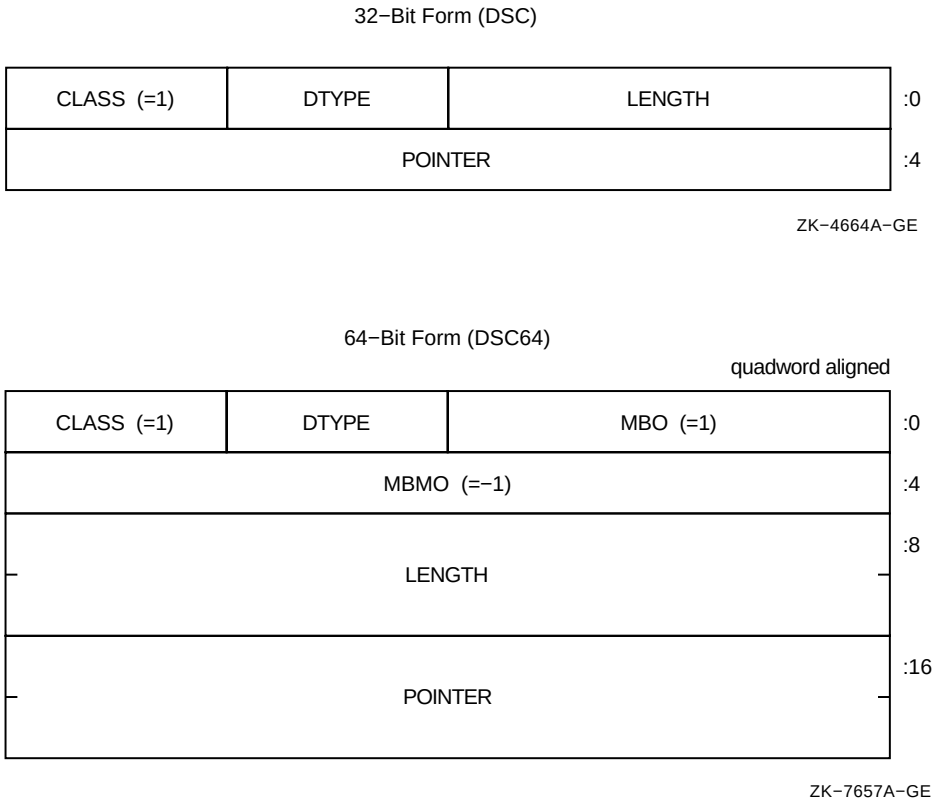


Table 5–3 Contents of the CLASS_S Descriptor

Symbol	Description
DSC$W_LENGTH DSC64$Q_LENGTH	Length of the data item in bytes, unless the DTYPE field contains the value 1 (aligned bit string) or 21 (packed-decimal string). Length of the data item is in bits for bit string. Length of the data item is the number of 4-bit digits (not including the sign) for a packed-decimal string.
DSC64$W_MBO	Must be 1. See Section 5.1.
DSC$B_DTYPE DSC64$B_DTYPE	A data-type code. Data-type codes are listed in Sections 4.1 and 4.2.

(continued on next page)

Table 5–3 (Cont.) Contents of the CLASS_S Descriptor

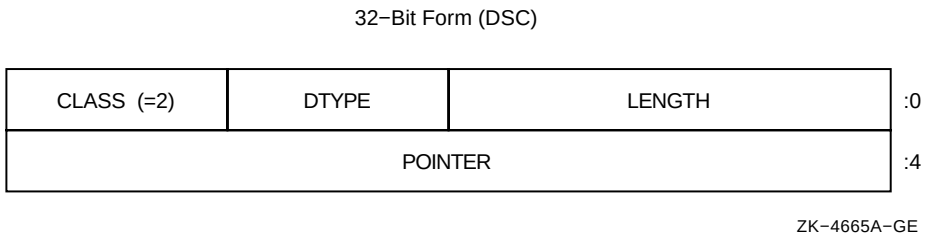
Symbol	Description
DSC$B_CLASS DSC64$B_CLASS	Defines the descriptor class code that must be equal to 1 for CLASS_S.
DSC$A_POINTER DSC64$PQ_POINTER	Address of first byte of data storage.
DSC64$L_MBMO	Must be -1. See Section 5.1.

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.

5.3 Dynamic String Descriptor (CLASS_D)

A class D descriptor is used for dynamically allocated strings. When a string is written, either the length field, pointer field, or both can be changed. The OpenVMS Run-Time Library provides procedures for changing fields. As an input parameter, this format is interchangeable with class 1 (CLASS_S). Figure 5–3 shows the format of a dynamic string descriptor. Table 5–4 describes the fields of the descriptor.

Figure 5–3 Dynamic String Descriptor Format



(continued on next page)

Figure 5–3 (Cont.) Dynamic String Descriptor Format

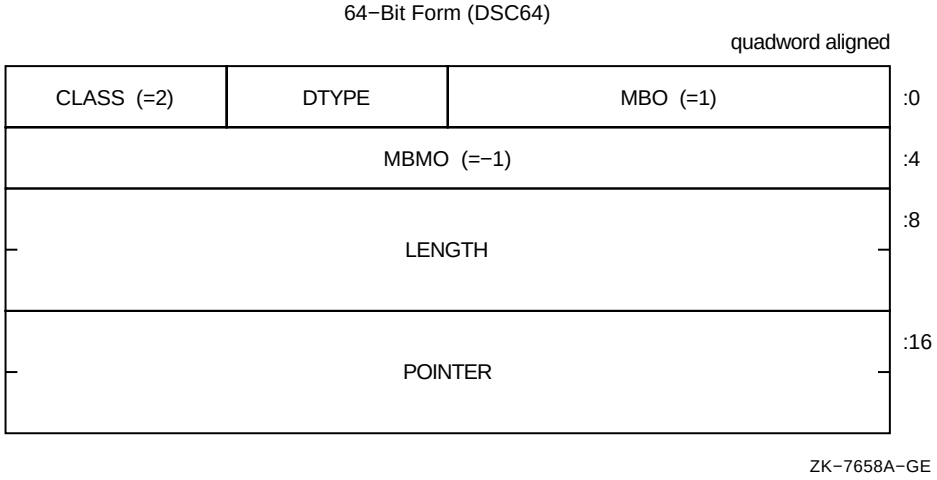


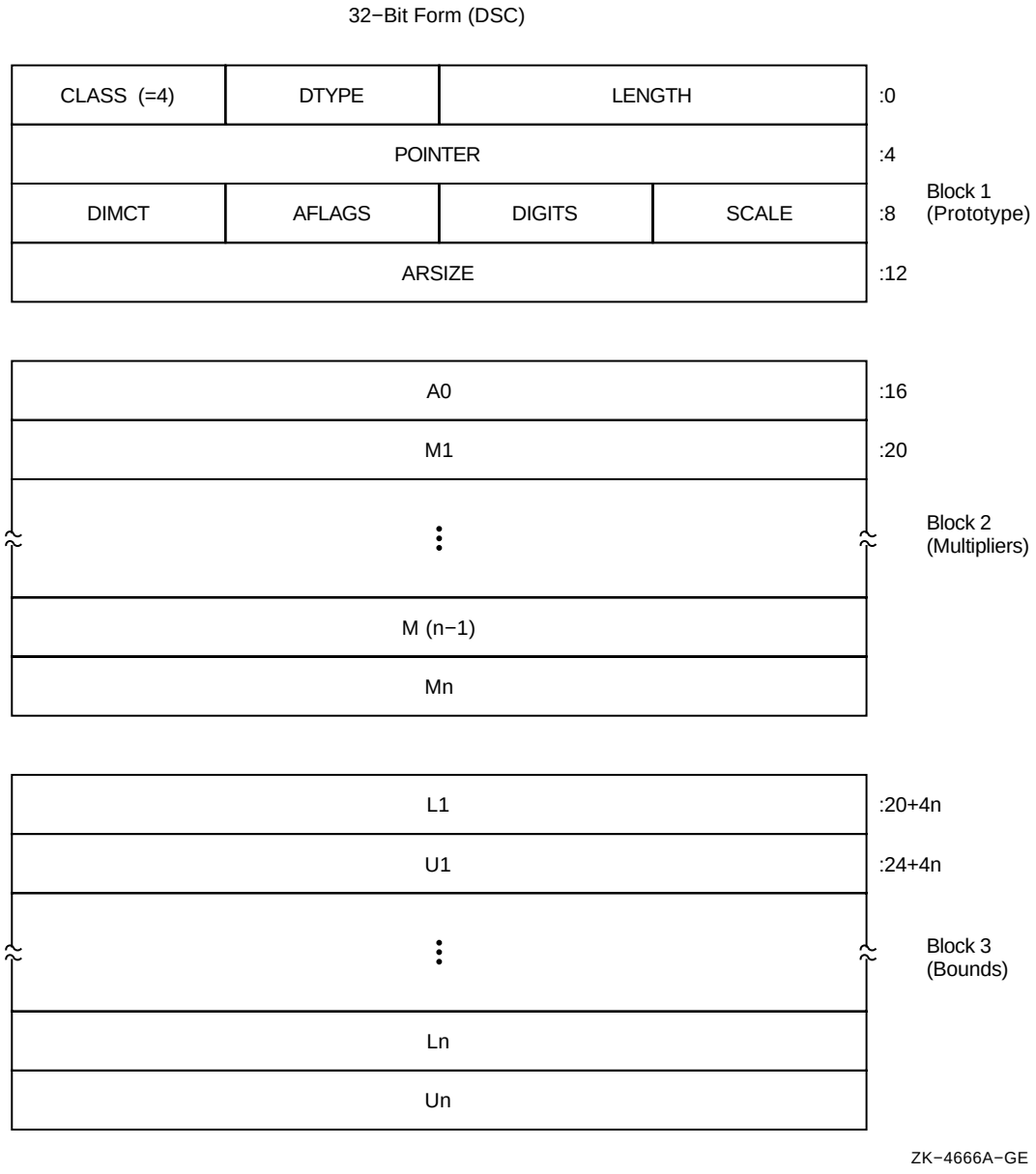
Table 5–4 Contents of the CLASS_D Descriptor

Symbol	Description
DSC$W_LENGTH DSC64$Q_LENGTH	Length of the data item in bytes, unless the DTYPE field contains the value 1 (aligned bit string) or 21 (packed-decimal string). Length of the data item is in bits for the bit string. Length of the data item is the number of 4-bit digits (not including the sign) for a packed-decimal string.
DSC64$W_MBO	Must be 1. See Section 5.1.
DSC$B_DTYPE DSC64$B_DTYPE	A data-type code. Data-type codes are listed in Sections 4.1 and 4.2.
DSC$B_CLASS DSC64$B_CLASS	Defines the descriptor class code that must be equal to 2 for CLASS_D.
DSC$A_POINTER DSC64$PQ_POINTER	Address of first byte of data storage.
DSC64$L_MBMO	Must be -1. See Section 5.1.

5.4 Array Descriptor (CLASS_A)

The array descriptor shown in Figure 5–4 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. Table 5–5 describes the fields of the descriptor.

Figure 5–4 Array Descriptor Format



(continued on next page)

Figure 5–4 (Cont.) Array Descriptor Format

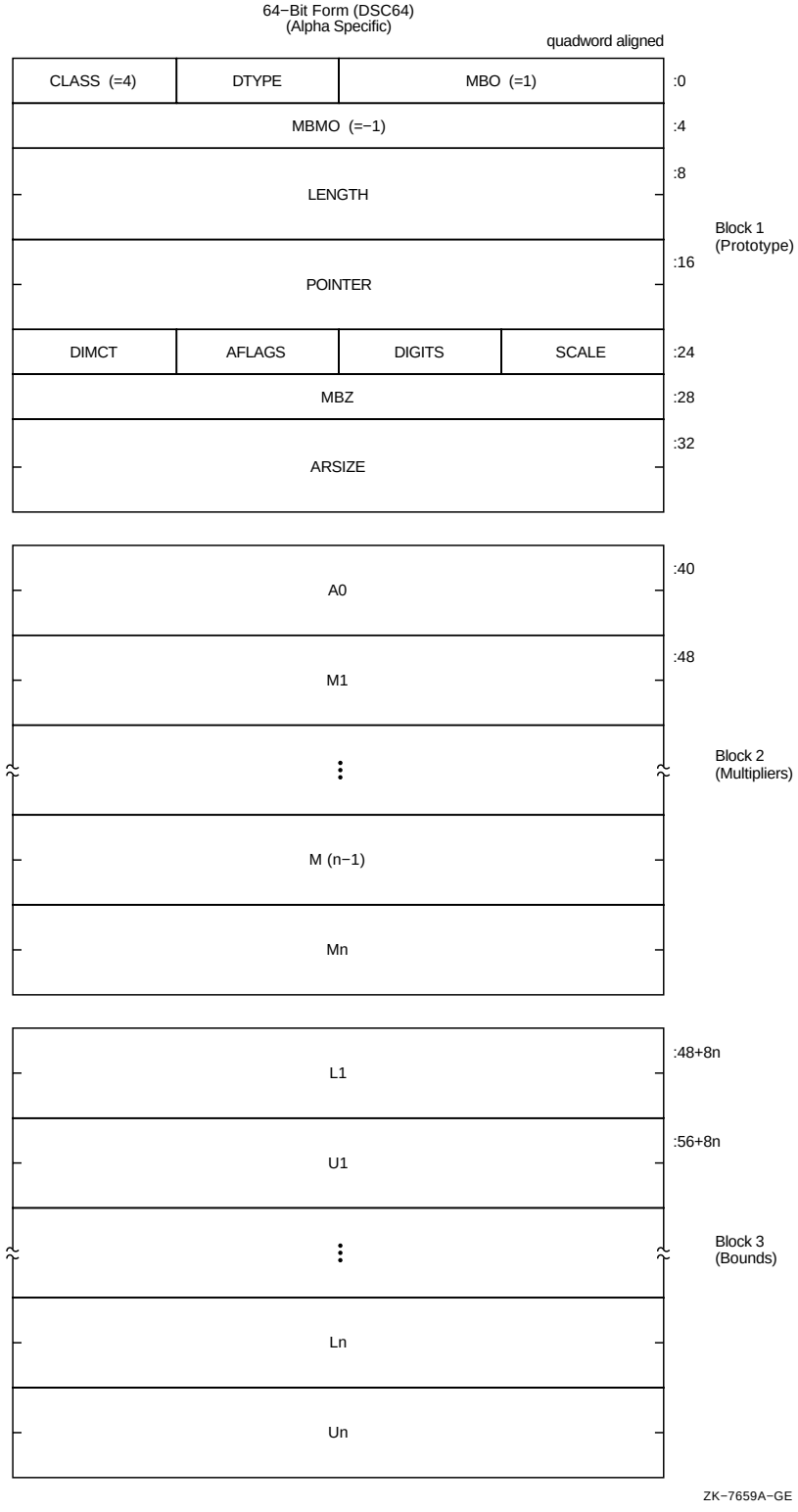


Table 5–5 Contents of the CLASS_A Descriptor

Symbol	Description
DSC$W_LENGTH DSC64$Q_LENGTH	Length of an array element in bytes, unless the DTYPE field contains the value 1 (aligned bit string) or 21 (packed-decimal string). Length of an array element is in bits for the bit string. Length of an array element is the number of 4-bit digits (not including the sign) for a packed-decimal string.
DSC64$W_MBO	Must be 1. See Section 5.1.
DSC$B_DTYPE DSC64$B_DTYPE	A data-type code. Data-type codes are listed in Sections 4.1 and 4.2.
DSC$B_CLASS DSC64$B_CLASS	Defines the descriptor class code that must be equal to 4 for CLASS_A.
DSC$A_POINTER DSC64$PQ_POINTER	Address of the first actual byte of data storage.
DSC64$L_MBMO	Must be -1. See Section 5.1.
DSC$B_SCALE DSC64$B_SCALE	Signed power-of-two or power-of-ten multiplier, as specified by FL_BINSCALE, to convert the internal form to external form. (See Section 5.6.)
DSC$B_DIGITS DSC64$B_DIGITS	If nonzero, the unsigned number of decimal digits in the internal representation. If 0, the number of digits can be computed based on LENGTH. This field should be 0 unless the TYPE field specifies a string data type that could contain numeric values.

(continued on next page)

Table 5–5 (Cont.) Contents of the CLASS_A Descriptor

Symbol	Description	
DSC$B_AFLAGS DSC64$B_AFLAGS	Array flag bits <23:16>:	
	Bits <18:16>	Reserved and must be 0.
	DSC$V_FL_BINSCALE DSC64$V_FL_BINSCALE	If set, the scale factor specified by SCALE is a signed power-of-two multiplier to convert the internal form to external form. If not set, SCALE specifies a signed power-of-ten multiplier. (See Section 5.6.)
	DSC$V_FL_REDIM DSC64$V_FL_REDIM	If set, the array can be redimensioned; that is, A0, M*i*, L*i*, and U*i* can be changed. The redimensioned array cannot exceed the size allocated to the array ARSIZE.
	DSC$V_FL_COLUMN DSC64$V_FL_COLUMN	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 (*n*th 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$V_FL_COEFF DSC64$V_FL_COEFF	If set, the multiplicative coefficients in block 2 are present. Must be set if FL_BOUNDS is set.
	DSC$V_FL_BOUNDS DSC64$V_FL_BOUNDS	If set, the bounds information in block 3 is present and requires that FL_COEFF be set.
DSC$B_DIMCT DSC64$B_DIMCT	Number of dimensions, *n*.	
DSC$L_ARSIZE DSC64$Q_ARSIZE	Total size of array (in bytes, unless the TYPE field contains the value 21; see the description for LENGTH). A redimensioned array can use less than the total size allocated.	
	For data type 1 (aligned bit string), LENGTH is in bits while ARSIZE is in bytes because the unit of length is bits, while the unit of allocation is aligned bytes.	
DSC$A_A0 DSC64$PQ_A0	Address of element A(0,0, . . . ,0). This need not be within the actual array. It is the same as POINTER for zero-origin arrays.	
DSC$L_M*i* DSC64$Q_M*i*	Addressing coefficients (M*i* = U*i* – L*i* + 1).	

(continued on next page)

Table 5–5 (Cont.) Contents of the CLASS_A Descriptor

Symbol	Description
DSC$L_L*i* DSC64$Q_L*i*	Lower bound (signed) of *i*th dimension.
DSC$L_U*i* DSC64$Q_U*i*	Upper bound (signed) of *i*th dimension.

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

Caution

Modification of the following formulas is required if DTYPE contains a 1 or 21, because LENGTH is given in bits or 4-bit digits rather than in bytes.

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

$$E = A_0 + I*LENGTH$$
$$= POINTER + [I - L_1]*LENGTH$$

The effective address, E, for element A(I_1,I_2) with FL_COLUMN clear:

$$E = A_0 + [I_1*M_2 + I_2]*LENGTH$$
$$= POINTER + [[I_1 - L_1]*M_2 + I_2 - L_2]*LENGTH$$

The effective address, E, for element A(I_1,I_2) with FL_COLUMN set:

$$E = A_0 + [I_2*M_1 + I_1]*LENGTH$$
$$= POINTER + [[I_2 - L_2]*M_1 + I_1 - L_1]*LENGTH$$

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

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

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

$$E = A_0 + [[[[...[I_n]*M_{n-1} + ...]*M_3 + I_3]*M_2 + I_2]*M_1 + I_1]*LENGTH$$
$$= POINTER + [[[[...[I_n - L_n]*M_{n-1} + ...]*M_3 + I_3 - L_3]*M_2 + I_2 - L_2]*M_1 + I_1 - L_1]*LENGTH$$

5.5 Procedure Argument Descriptor (CLASS_P)

A descriptor for a procedure argument identifies a procedure and its result data type, if any.

On VAX processors, the descriptor for a procedure argument specifies its entry address and function value data type. On Alpha processors, the procedure argument descriptor is a pointer to the procedure descriptor, which is described in Section 3.4. Figure 5–5 shows the format of a procedure argument descriptor. Table 5–6 describes the fields of the descriptor.

Figure 5–5 Procedure Argument Descriptor Format

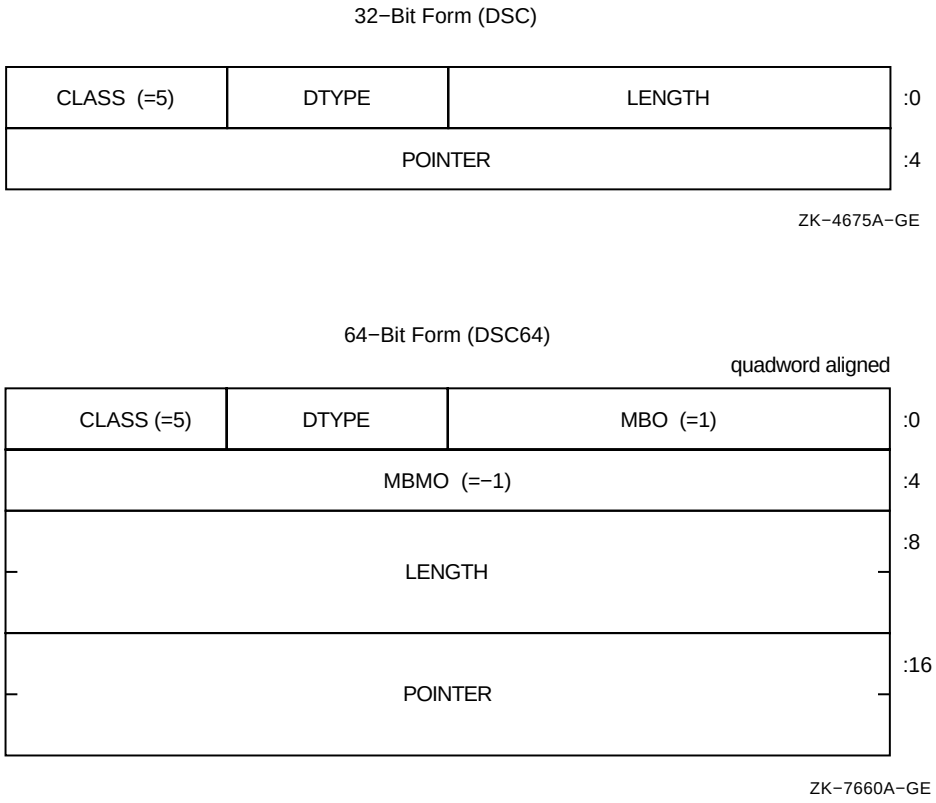


Table 5–6 Contents of the CLASS_P Descriptor

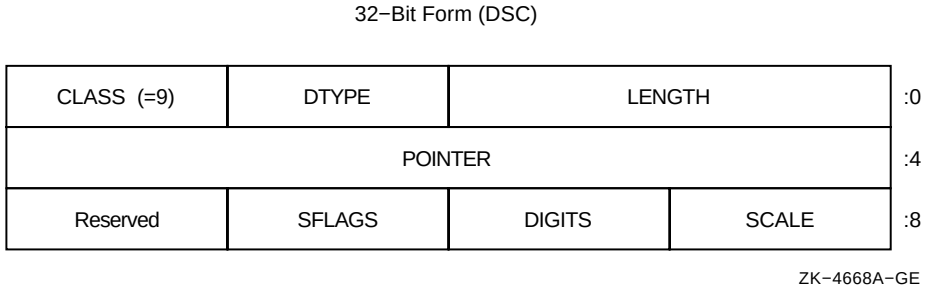
Symbol	Description
DSC$W_LENGTH DSC64$Q_LENGTH	Length associated with the function value, or 0 if no function value is returned.
DSC64$W_MBO	Must be 1. See Section 5.1.
DSC$B_DTYPE DSC64$B_DTYPE	Function value data-type code. Data-type codes are listed in Sections 4.1 and 4.2.
DSC$B_CLASS DSC64$B_CLASS	Defines the descriptor class code that must be equal to 5 for CLASS_P.
DSC$A_POINTER DSC64$PQ_POINTER	Address of entry mask to the procedure for VAX environments. Address of the procedure descriptor of the procedure for Alpha environments.
DSC64$L_MBMO	Must be -1. See Section 5.1.

Procedures return a function value as described in Section 2.5 for VAX or Section 3.8.7 for Alpha.

5.6 Decimal String Descriptor (CLASS_SD)

Figure 5–6 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. Table 5–7 describes the fields of the descriptor.

Figure 5–6 Decimal String Descriptor Format



(continued on next page)

Figure 5–6 (Cont.) Decimal String Descriptor Format

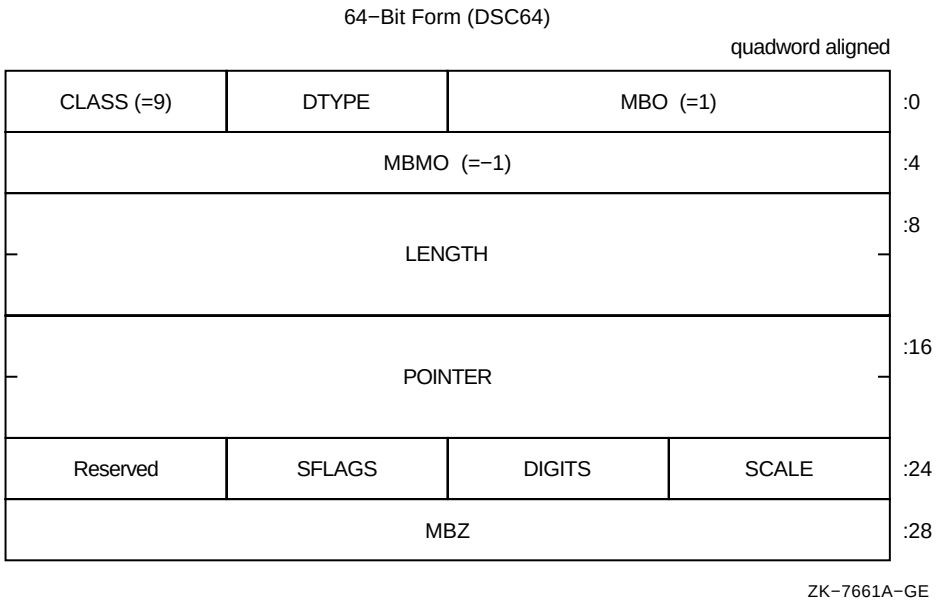


Table 5–7 Contents of the CLASS_SD Descriptor

Symbol	Description
DSC$W_LENGTH DSC64$Q_LENGTH	Length of the data item in bytes, unless the DTYPE field contains the value 1 (aligned bit string) or 21 (packed-decimal string). Length of the data item is in bits for the bit string. Length of the data item is the number of 4-bit digits (not including the sign) for packed-decimal string.
DSC64$W_MBO	Must be 1. See Section 5.1.
DSC$B_DTYPE DSC64$B_DTYPE	A data-type code. Data-type codes are listed in Sections 4.1 and 4.2.
DSC$B_CLASS DSC64$B_CLASS	Defines the descriptor class code that must be equal to 9 for CLASS_SD.
DSC$A_POINTER DSC64$PQ_POINTER	Address of the first byte of data storage.
DSC64$L_MBMO	Must be -1. See Section 5.1.
DSC$B_SCALE DSC64$B_SCALE	Signed power-of-two or power-of-ten multiplier, as specified by FL_BINSCALE, to convert the internal form to external form. (See examples in Table 5–8.)

(continued on next page)

Table 5–7 (Cont.) Contents of the CLASS_SD Descriptor

Symbol	Description	
DSC$B_DIGITS DSC64$B_DIGITS	If nonzero, the unsigned number of decimal digits in the internal representation. If 0, the number of digits can be computed based on LENGTH. This field should be 0 unless the TYPE field specifies a string data type that could contain numeric values.	
DSC$B_SFLAGS DSC64$B_SFLAGS	Scalar flag bits <23:16>:	
	Bits <18:16>	Reserved and must be 0.
	DSC$V_FL_BINSCALE DSC64$V_FL_BINSCALE	If set, the scale factor specified by SCALE is a signed power-of-two multiplier to convert the internal form to external form. If not set, SCALE specifies a signed power-of-ten multiplier. (See examples in Table 5–8.)
	Bit <23:20>	Reserved and must be 0.

Examples of SCALE and FL_BINSCALE interpretation are presented in Table 5–8.

Table 5–8 Internal-to-External BINSCALE Conversion Examples

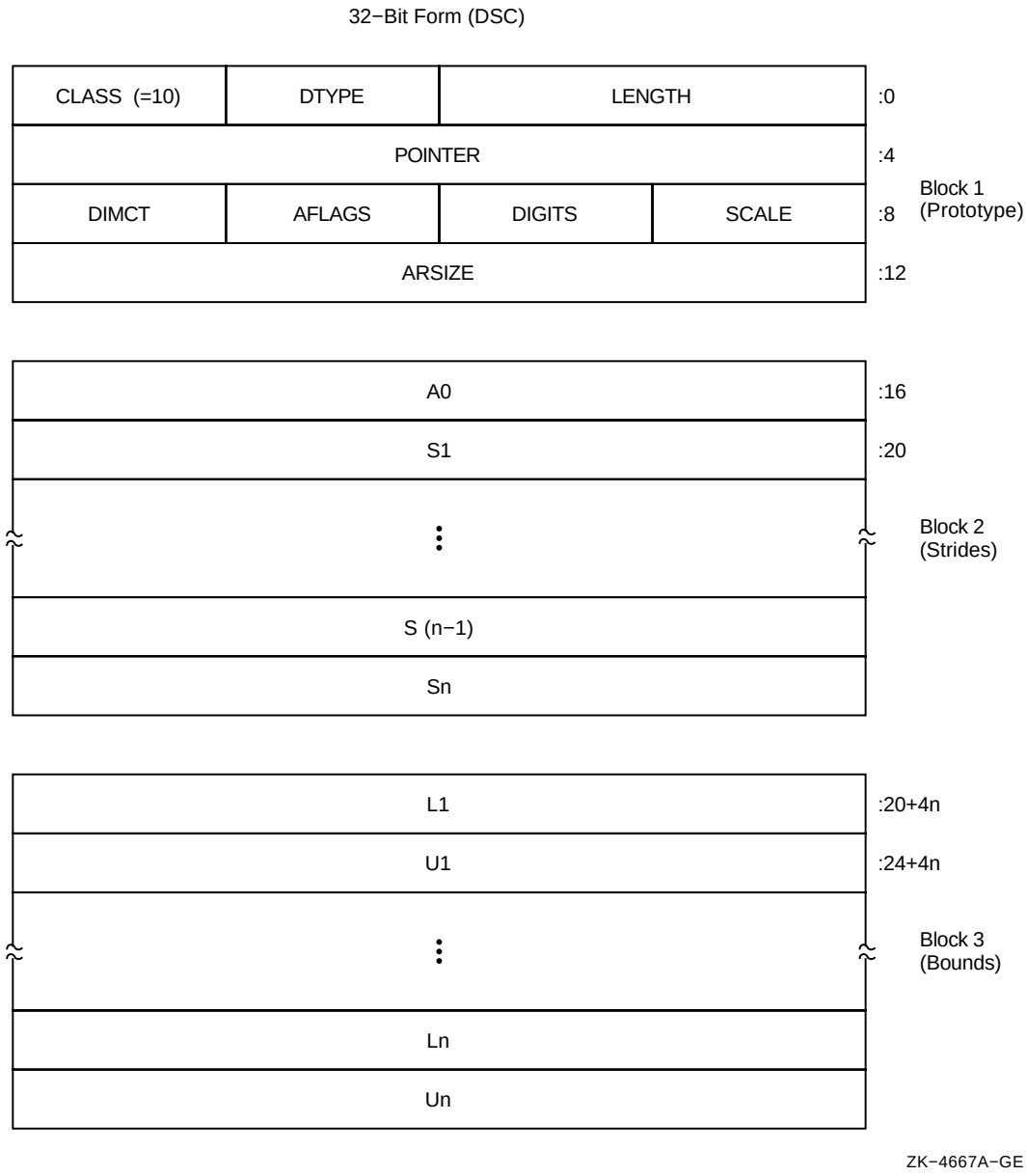
Internal Value	SCALE	FL_BINSCALE	External Value
123	+1	0	1230
123	+1	1	246
200	–2	0	2
200	–2	1	50

5.7 Noncontiguous Array Descriptor (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**. You can align elements by row or column, because the accessing algorithm in the called procedure handles both alignments.

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 5.4, for language processors that always allocate contiguous arrays. Figure 5–7 shows the format of a noncontiguous array descriptor, which consists of three contiguous blocks. Table 5–9 describes the fields of the descriptor.

Figure 5–7 Noncontiguous Array Descriptor Format



(continued on next page)

Figure 5–7 (Cont.) Noncontiguous Array Descriptor Format

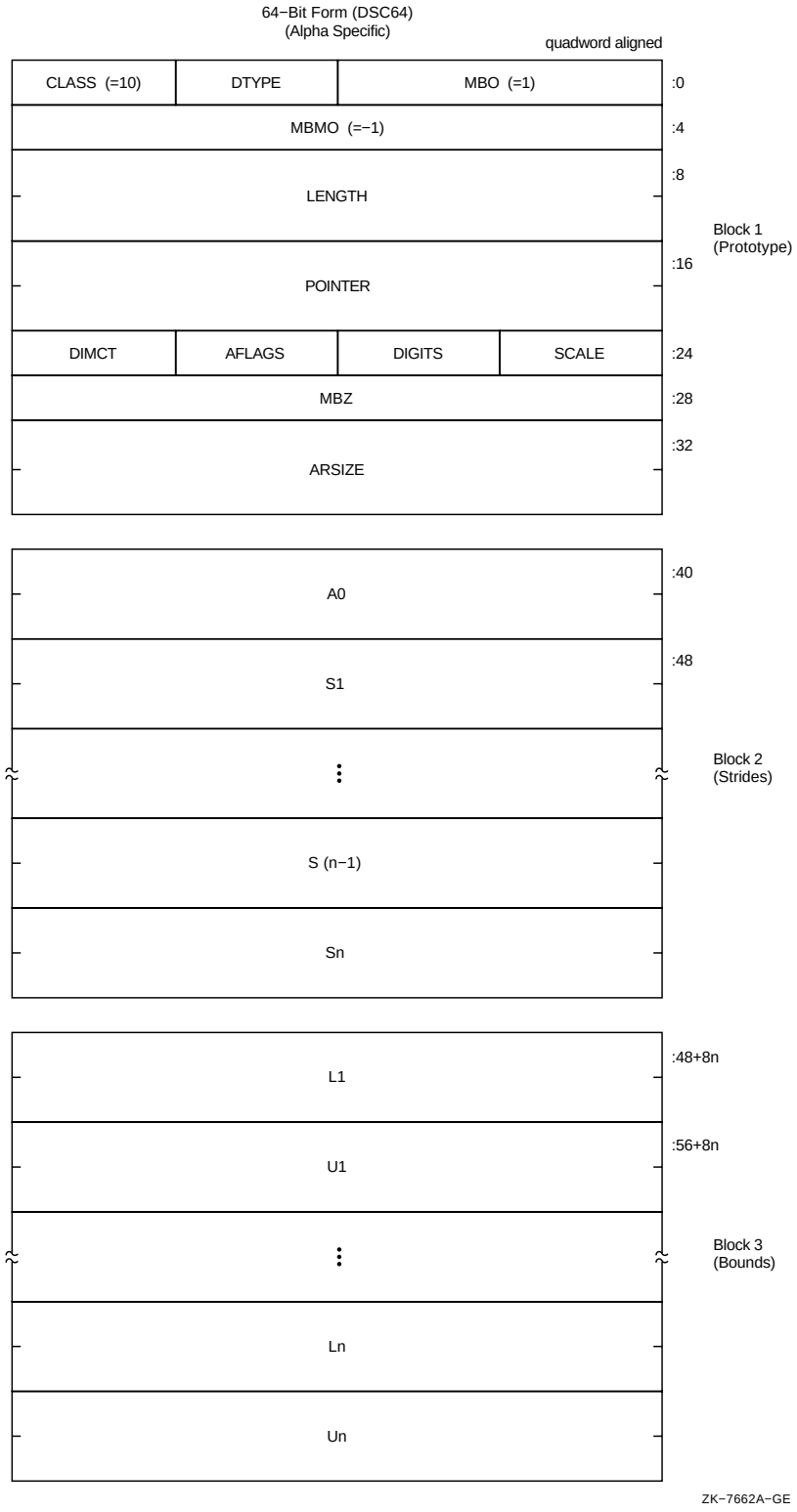


Table 5–9 Contents of the CLASS_NCA Descriptor

Symbol	Description
DSC$W_LENGTH DSC64$Q_LENGTH	Length of an array element in bytes, unless the DTYPE field contains the value 1 (aligned bit string) or 21 (packed-decimal string). Length of an array element is in bits for the bit string. Length of an array element is the number of 4-bit digits (not including the sign) for a packed-decimal string.
DSC64$W_MBO	Must be 1. See Section 5.1.
DSC$B_DTYPE DSC64$B_DTYPE	A data-type code. Data-type codes are listed in Sections 4.1 and 4.2.
DSC$B_CLASS	Defines the descriptor class code that must be equal to 10 for CLASS_NCA.
DSC$A_POINTER DSC64$PQ_POINTER	Address of first actual byte of data storage.
DSC64$L_MBMO	Must be -1. See Section 5.1.
DSC$B_SCALE DSC64$B_SCALE	Signed power-of-two or power-of-ten multiplier, as specified by FL_BINSCALE, to convert the internal form to external form. (See Section 5.6.)
DSC$B_DIGITS DSC64$B_DIGITS	If nonzero, the unsigned number of decimal digits in the internal representation. If 0, the number of digits can be computed based on LENGTH. This field should be 0 unless the TYPE field specifies a string data type that could contain numeric values.

(continued on next page)

Table 5–9 (Cont.) Contents of the CLASS_NCA Descriptor

Symbol	Description	
DSC$B_AFLAGS DSC64$B_AFLAGS	Array flag bits <23:16>:	
	Bits <18:16>	Reserved to Compaq. Must be 0.
	DSC$V_FL_BINSCALE DSC64$V_FL_BINSCALE	If set, the scale factor specified by SCALE is a signed power-of-two multiplier to convert the internal form to external form. If not set, SCALE specifies a signed power-of-ten multiplier. (See Section 5.6.)
	DSC$V_FL_REDIM DSC64$V_FL_REDIM	Must be 0.
	DSC$V_FL_UNALLOC DSC64$V_FL_UNALLOC	If set, the storage for the array described by this descriptor has not been allocated; the POINTER field must contain 0. If not set, storage for the array described by this descriptor has been allocated; the POINTER field may or may not be 0, depending on the bounds of the array. (If the POINTER field contains a nonzero value, then this flag must not be set.)
	DSC$V_FL_NODEALLOC	If set, the storage for the array described by this descriptor must not be deallocated. (The POINTER and other fields of this descriptor may be cleared or otherwise set to eliminate access to the described storage, but the storage itself belongs to some other descriptor which must be used to deallocate that storage.)
	Bit <23:23>	Reserved to Compaq. Must be 0.
DSC$B_DIMCT DSC64$B_DIMCT	Number of dimensions, *n*.	
DSC$L_ARSIZE DSC64$Q_ARSIZE	If the elements are contiguous, ARSIZE is the total size of the array (in bytes, unless the DTYPE field contains the value 21; see the description of 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), LENGTH is in bits while ARSIZE is in bytes because the unit of length is in bits while the unit of allocation is in bytes.	

(continued on next page)

Table 5–9 (Cont.) Contents of the CLASS_NCA Descriptor

Symbol	Description
DSC$A_A0 DSC64$PQ_A0	Address of element A(0,0, . . . ,0). This need not be within the actual array. It is the same as POINTER for zero-origin arrays. A0 = POINTER − (S_1*L_1 + S_2*L_2 + . . . + S_n*L_n)
DSC$L_Si DSC64$Q_Si	Stride of the *i*th dimension. The difference between the addresses of successive elements of the *i*th dimension.
DSC$L_L*i* DSC64$Q_L*i*	Lower bound (signed) of the *i*th dimension.
DSC$L_U*i* DSC64$Q_U*i*	Upper bound (signed) of the *i*th dimension.

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

Caution

Modification of the following formulas is required if DTYPE equals 1 or 21 because LENGTH is given in bits or 4-bit digits rather than bytes.

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

$$E = A_0 + S_1*I$$
$$= POINTER + S_1*[I - L_1]$$

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

$$E = A_0 + S_1*I_1 + S_2*I_2$$
$$= POINTER + S_1*[I_1 - L_1] + S_2*[I_2 - L_2]$$

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

$$E = A_0 + S_1*I_1 + . . . + S_n*I_n$$
$$= POINTER + S_1*[I_1 - L_1] + . . . + S_n*[I_n - L_n]$$

5.8 Varying String Descriptor (CLASS_VS)

A class VS descriptor is used for varying string data types (see Section 4.5).

As an input parameter, this format is not interchangeable with class 1 (CLASS_S) or with class 2 (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 5–8 shows the format of a varying string descriptor. Table 5–10 describes the fields of the descriptor.

Figure 5–8 Varying String Descriptor Format

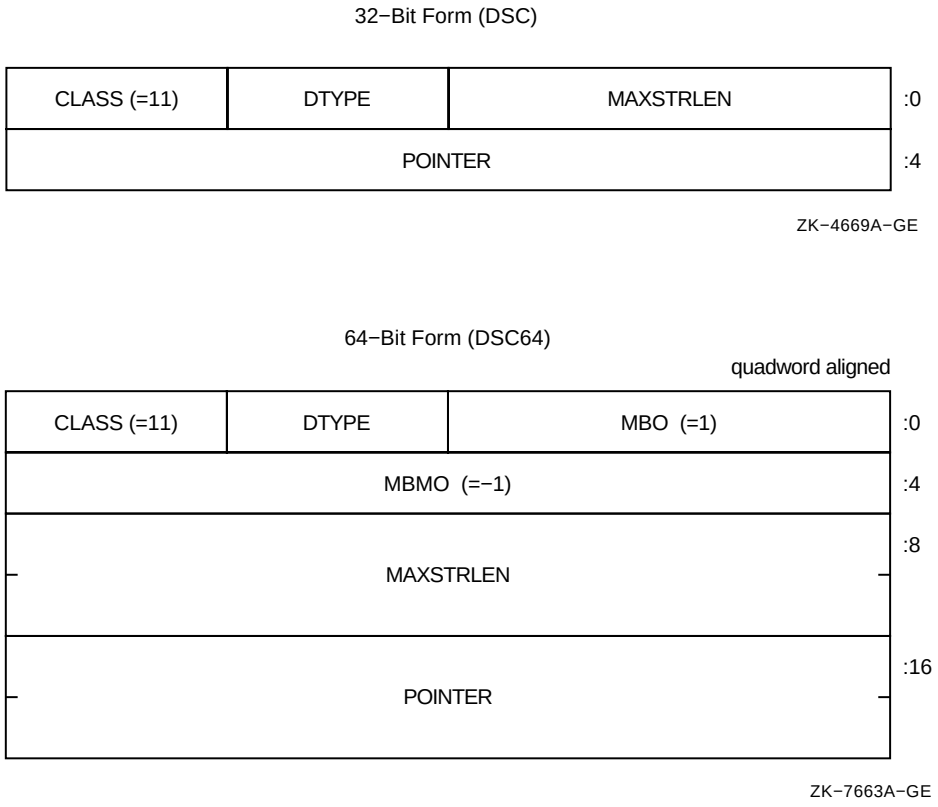
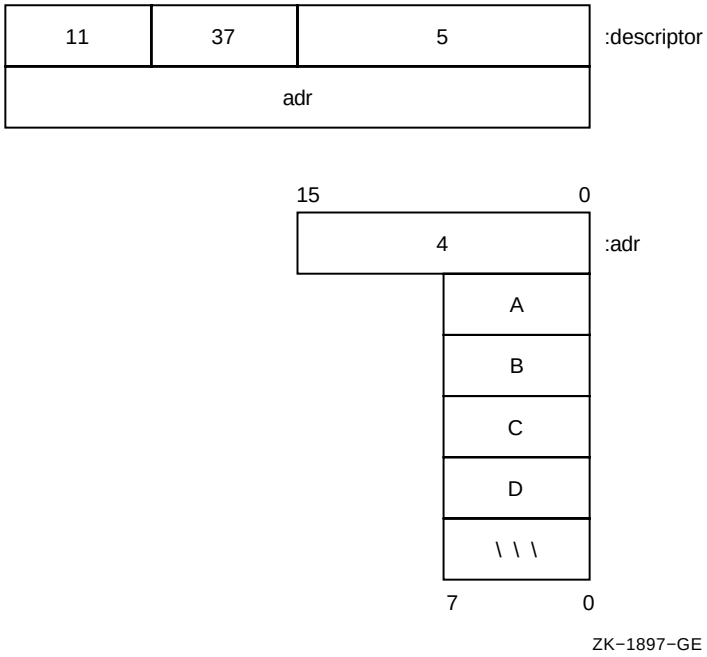


Table 5–10 Contents of the CLASS_VS Descriptor

Symbol	Description
DSC$W_MAXSTRLEN DSC64$Q_MAXSTRLEN	Maximum length of the BODY field of the varying string in bytes in the range 0 to $2^{16} - 1$.
DSC64$W_MBO	Must be 1. See Section 5.1.
DSC$B_DTYPE DSC64$B_DTYPE	A data type code that has the value 37, which specifies the varying character string data type (see Sections 4.2 and 4.5). The use of other data types is reserved to Compaq.
DSC$B_CLASS DSC64$B_CLASS	Defines the descriptor class code that must be equal to 11 for CLASS_VS.
DSC$A_POINTER DSC64$PQ_POINTER	Address of the first field (CURLEN) of the varying string.
DSC64$L_MBMO	Must be -1. See Section 5.1.

Figure 5–9 illustrates the use of a 32-bit varying string descriptor to present a variable that is capable of holding a string value of up to five characters in length and that is currently holding the string value ABCD. As shown in the figure, MAXSTRLEN contains five, CURLEN contains four, string is currently ABCD, and the remaining byte is currently undefined.

Figure 5–9 Varying String Descriptor with Character String Data Type

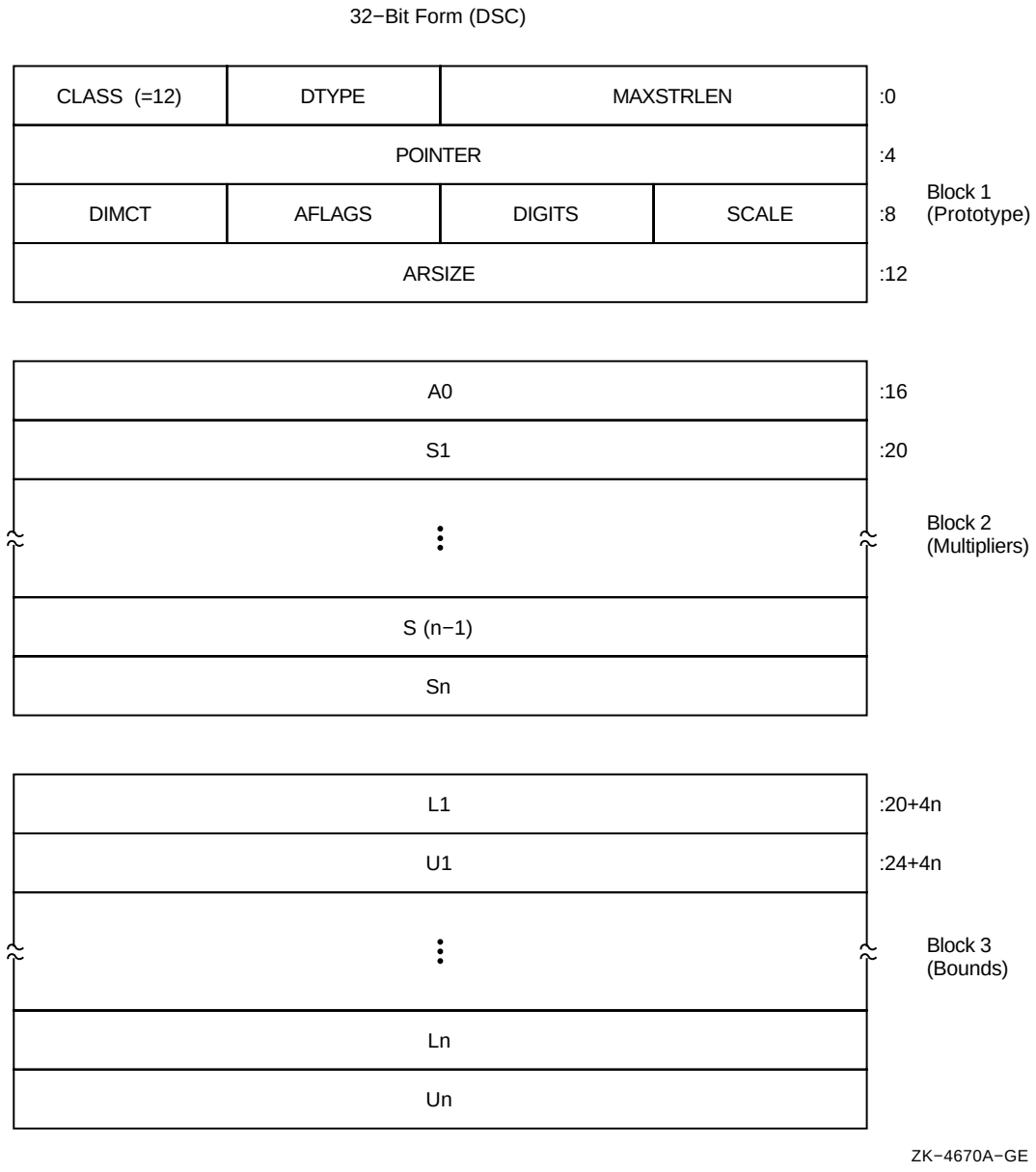


5.9 Varying String Array Descriptor (CLASS_VSA)

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 of the varying string data type (see Section 4.5).

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 5–10 shows the format of a varying string array descriptor. Table 5–11 describes the fields of the descriptor.

Figure 5–10 Varying String Array Descriptor Format



(continued on next page)

Figure 5–10 (Cont.) Varying String Array Descriptor Format

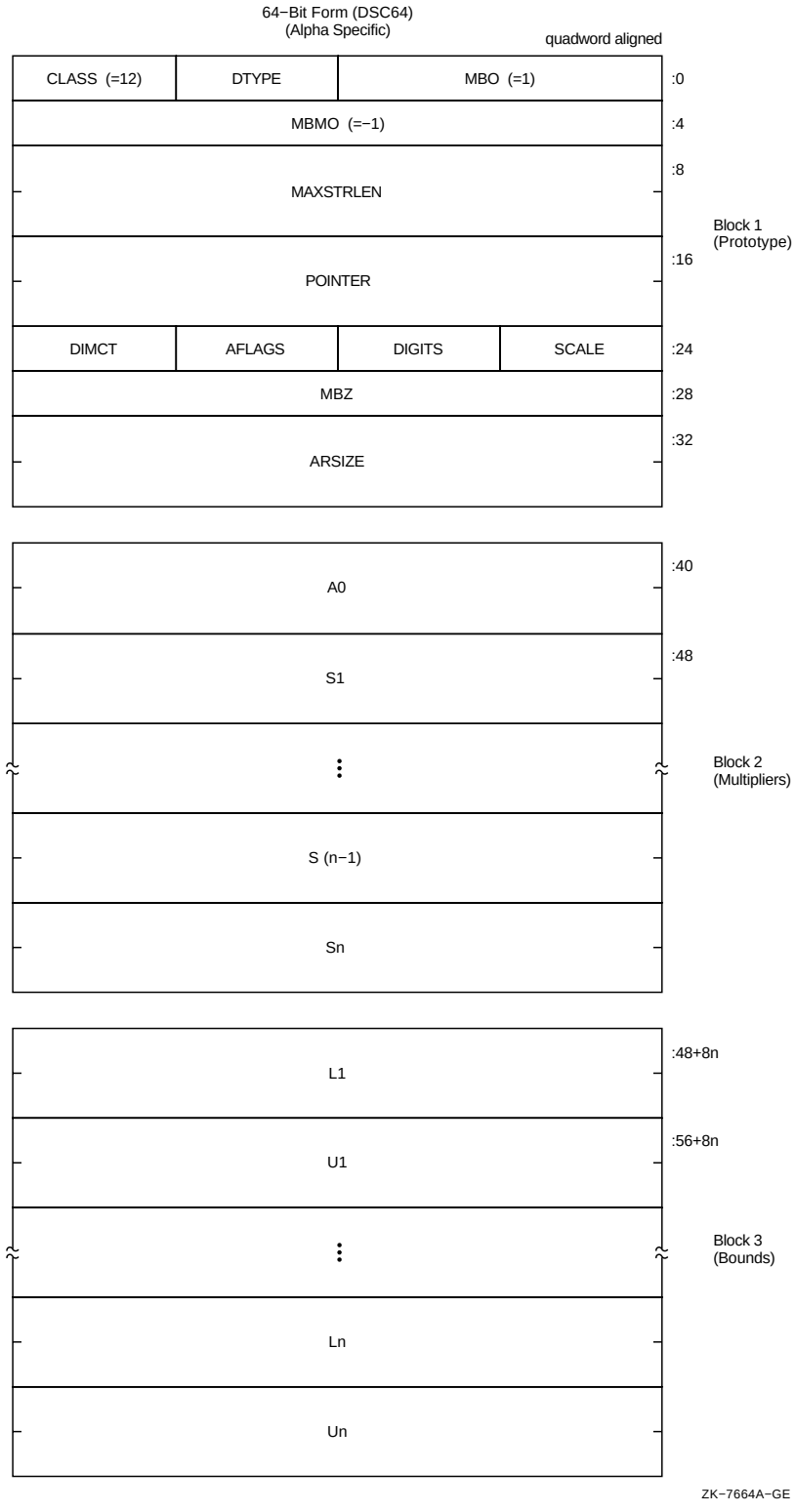


Table 5–11 Contents of the CLASS_VSA Descriptor

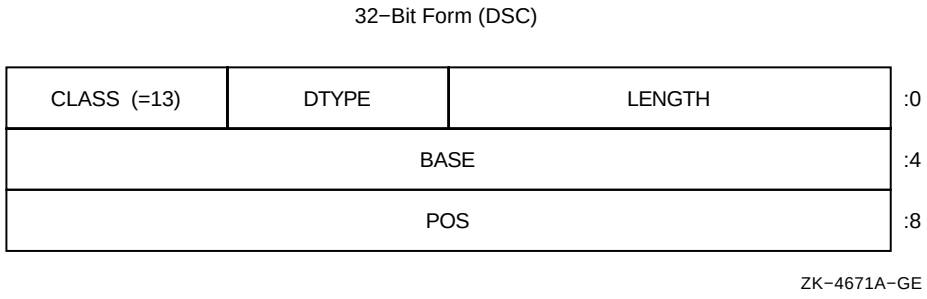
Symbol	Description
DSC$W_MAXSTRLEN DSC64$Q_MAXSTRLEN	Maximum length of the BODY field of an array element in bytes in the range 0 to $2^{16} - 1$.
DSC64$W_MBO	Must be 1. See Section 5.1.
DSC$B_DTYPE DSC64$B_DTYPE	A data-type code that has the value 37, which specifies the varying character string data type (see Sections 4.2 and 4.5). The use of other data types is reserved to Compaq.
DSC$B_CLASS DSC64$B_CLASS	Defines the descriptor class code that must be equal to 12 for CLASS_VSA.
DSC$A_POINTER DSC64$PQ_POINTER	Address of the first actual byte of data storage.
DSC64$L_MBMO	Must be -1. See Section 5.1.

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.

5.10 Unaligned Bit String Descriptor (CLASS_UBS)

A descriptor is used to pass an unaligned bit string (DSC$K_DTYPE_VU) that starts and ends on an arbitrary bit boundary. The descriptor provides two components: a base address and a signed relative bit position. Figure 5–11 shows the format of an unaligned bit string descriptor. Table 5–12 describes the fields of the descriptor.

Figure 5–11 Unaligned Bit String Descriptor Format



(continued on next page)

Figure 5–11 (Cont.) Unaligned Bit String Descriptor Format

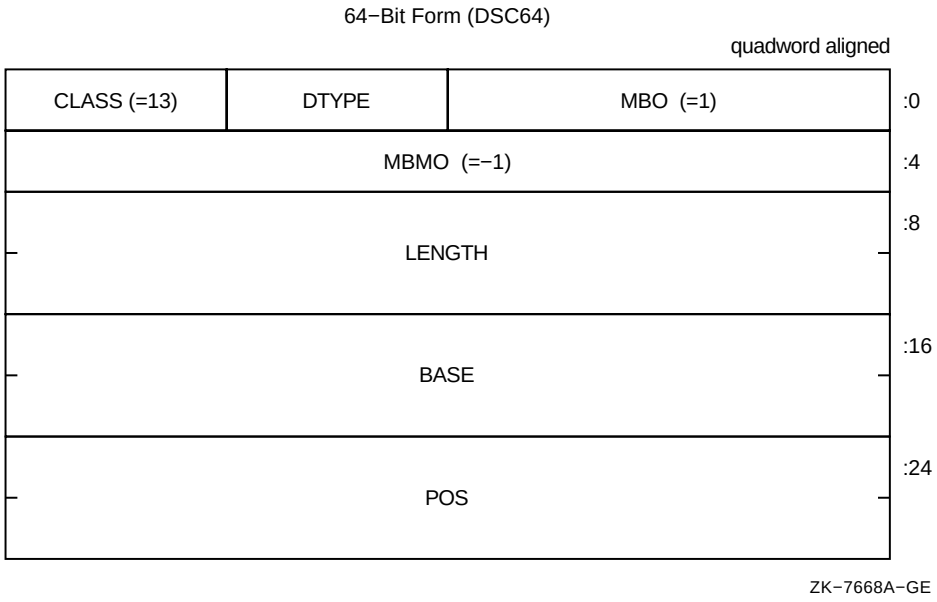


Table 5–12 Contents of the CLASS_UBS Descriptor

Symbol	Description
DSC$W_LENGTH DSC64$Q_LENGTH	Length of data item in bits.
DSC64$W_MBO	Must be 1. See Section 5.1.
DSC$B_DTYPE DSC64$B_DTYPE	A data-type code that has the value 34, which specifies the unaligned bit string data type (see Sections 4.1 and 4.2). The use of other data types is reserved to Compaq.
DSC$B_CLASS DSC64$B_CLASS	Defines the descriptor class code that must be equal to 13 for CLASS_UBS.
DSC$A_BASE DSC64$PQ_BASE	Base of the address relative to which the signed relative bit position, POS, is used to locate the bit string. The base address need not be the first actual byte of data storage.
DSC64$L_MBMO	Must be -1. See Section 5.1.
DSC$L_POS DSC64$Q_POS	Relative bit position with respect to BASE of the first bit of unaligned bit string.

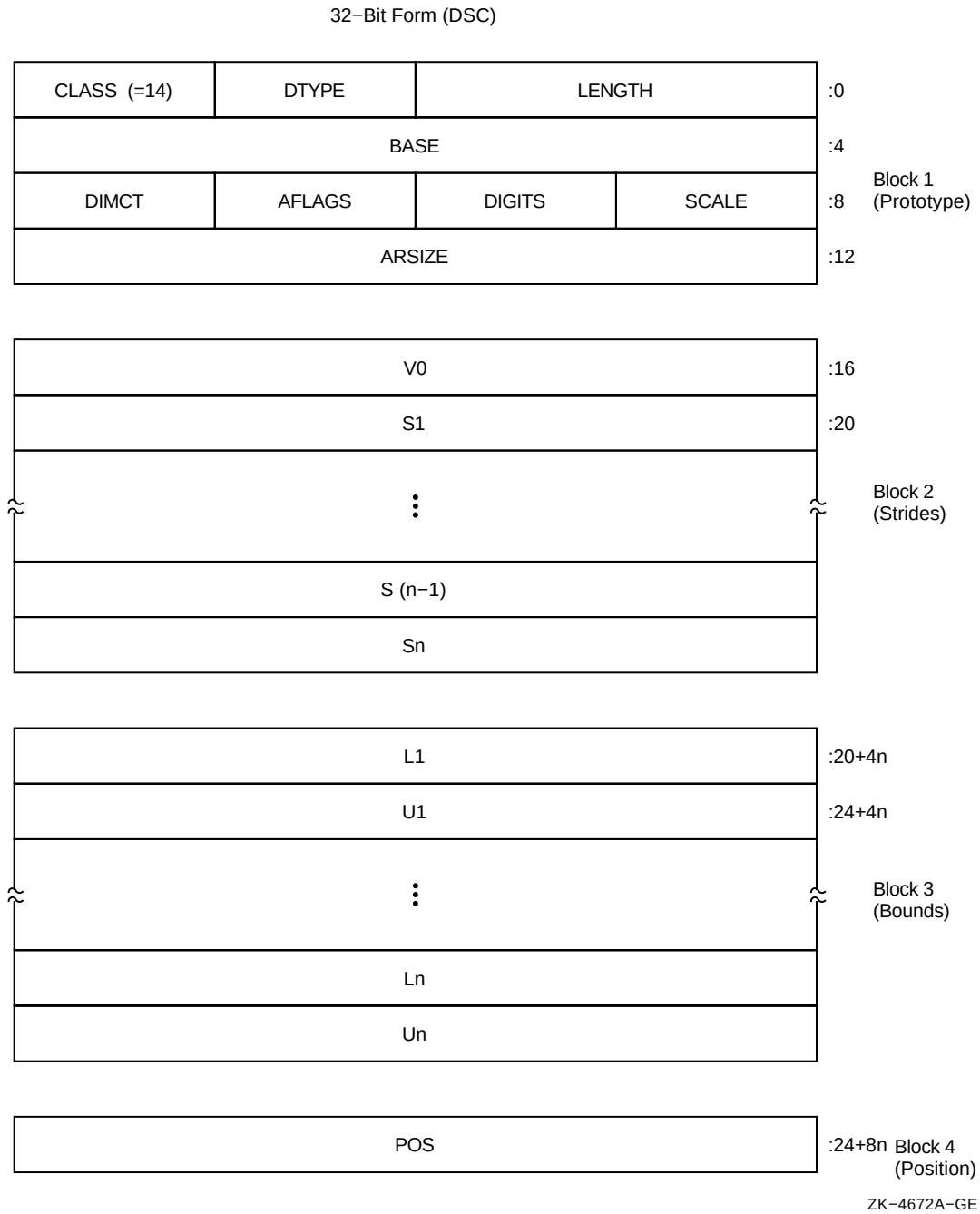
5.11 Unaligned Bit Array Descriptor (CLASS_UBA)

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 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 by using the VAX variable bit field instructions. Therefore, the descriptor provides two components: a byte address, 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 5–12 shows the format of an unaligned bit array descriptor. Table 5–13 describes the fields of the descriptor.

Figure 5–12 Unaligned Bit Array Descriptor Format



(continued on next page)

Figure 5–12 (Cont.) Unaligned Bit Array Descriptor Format

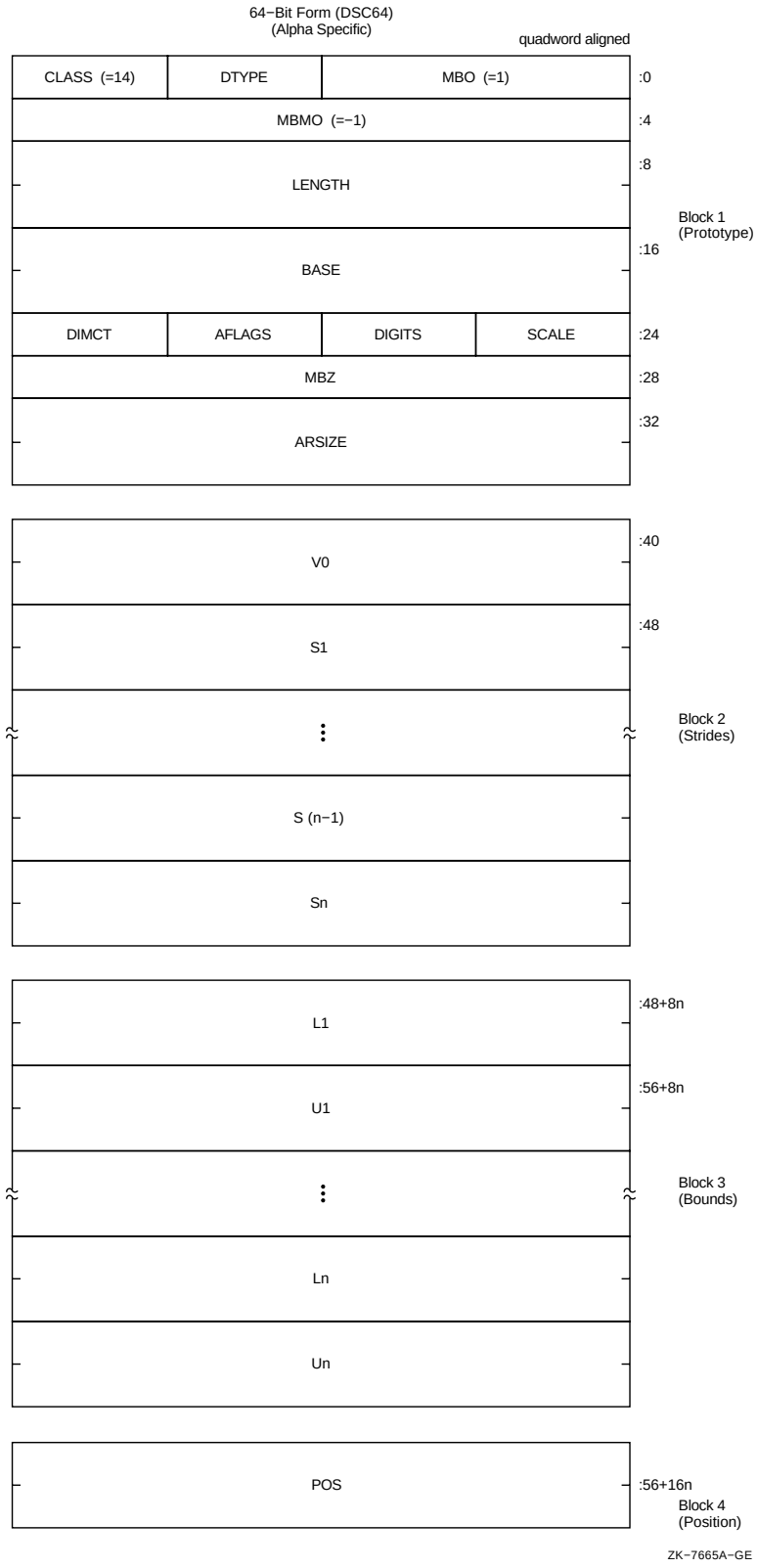


Table 5–13 Contents of the CLASS_UBA Descriptor

Symbol	Description	
DSC$W_LENGTH DSC64$Q_LENGTH	Length of an array element in bits.	
DSC64$W_MBO	Must be 1. See Section 5.1.	
DSC$B_DTYPE DSC64$B_DTYPE	A data-type code that must have the value 34, which specifies the unaligned bit string data type (see Sections 4.1 and 4.2). The use of other data types is reserved to Compaq.	
DSC$B_CLASS DSC64$B_CLASS	Defines the descriptor class code that must be equal to 14 for CLASS_UBA.	
DSC$A_BASE DSC64$PQ_BASE	Base address relative to the effective bit offset, EB, that is used to locate elements of the array. The base address need not be the first actual byte of data storage.	
DSC64$L_MBMO	Must be -1. See Section 5.1.	
DSC$B_SCALE DSC64$B_SCALE	Reserved to Compaq. Must be 0.	
DSC$B_DIGITS DSC64$B_DIGITS	If nonzero, the unsigned number of decimal digits in the internal representation. If 0, the number of digits can be computed based on LENGTH. This field should be 0 unless the TYPE field specifies a string data type that could contain numeric values.	
DSC$B_AFLAGS DSC64$B_AFLAGS	Array flag bits <23:16>:	
	Bits <18:16>	Reserved to Compaq. Must be 0.
	DSC$V_FL_BINSCALE DSC64$V_FL_BINSCALE	Must be 0.
	DSC$V_FL_REDIM DSC64$V_FL_REDIM	Must be 0.
	Bits <23:21>	Reserved to Compaq. Must be 0.
DSC$B_DIMCT DSC64$B_DIMCT	Number of dimensions, *n*.	
DSC$L_ARSIZE DSC64$Q_ARSIZE	If the elements are 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$L_V0 DSC64$Q_V0	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$L_Si DSC64$Q_Si	Stride of the *i*th dimension. The difference between the bit (not byte) addresses of successive elements of the *i*th dimension.	
DSC$L_L*i* DSC64$Q_L*i*	Lower bound (signed) of the *i*th dimension.	
DSC$L_U*i* DSC64$Q_U*i*	Upper bound (signed) of the *i*th dimension.	
DSC$L_POS DSC64$Q_POS	Relative bit position with respect to BASE of the first actual bit of the array, that is, element A(L_1, . . . ,L_n).	

The following formulas specify the signed effective bit offset, EB, of an array element:

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

$$EB = V_0 + S_1*I_1$$
$$= POS + S_1*[I_1 - L_1]$$

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

$$EB = V_0 + S_1*I_1 + S_2*I_2$$
$$= POS + S_1*[I_1 - L_1] + S_2*[I_2 - L_2]$$

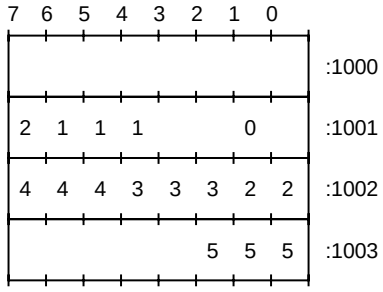
The signed effective bit offset, EB, of $A(I_1, \ldots, I_n)$:

$$EB = V_0 + S_1*I_1 + \ldots + S_n*I_n$$
$$= POS + S_1*[I_1 - L_1] + \ldots + S_n*[I_n - L_n]$$

Note that EB is computed ignoring integer overflow.

On VAX systems, EB is used as the position operand, and the content of BASE is used as the base address operand in the VAX variable-length bit field instructions. Therefore, BASE must specify a byte within 2^{28} bytes of all bytes of storage in the bit array.

For example, consider a single-origin, one-dimensional, five-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.



ZK–1901–GE

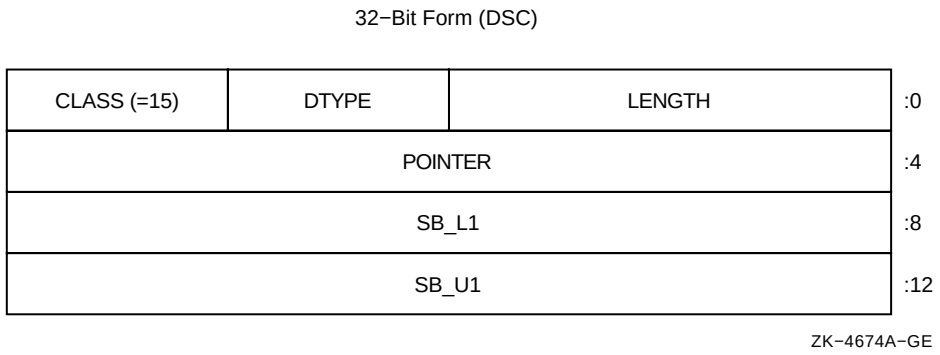
The following dependent field values occur in the descriptor:

$$POS = 12$$
$$V_0 = 12 - 3*1 = 9$$

5.12 String with Bounds Descriptor (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 5–13 shows the format of a string with bounds descriptor. Table 5–14 describes the fields of the descriptor.

Figure 5–13 String with Bounds Descriptor Format



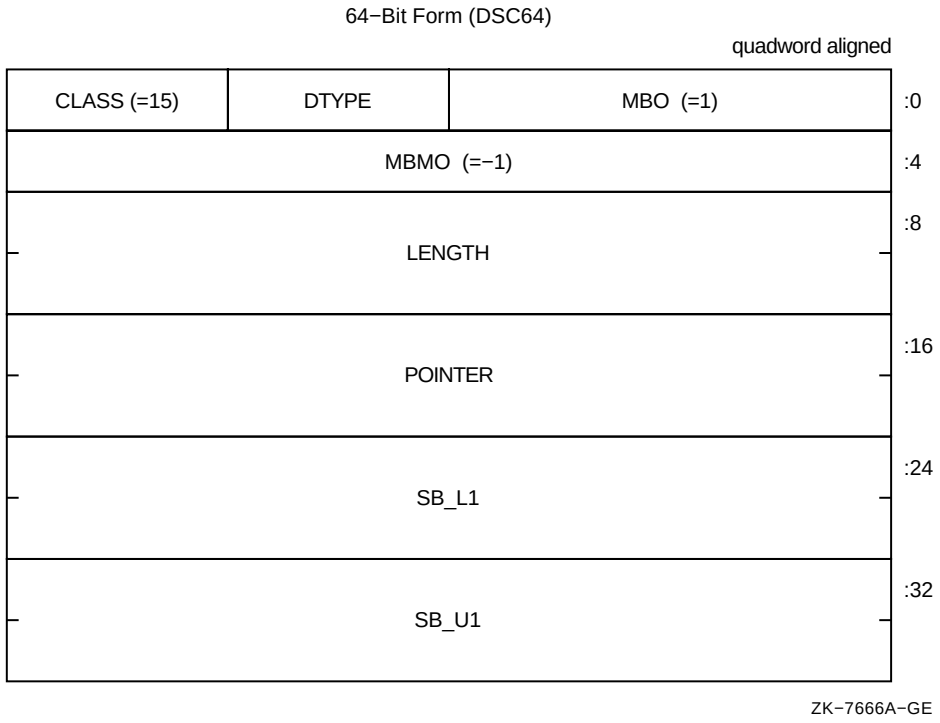


Table 5–14 Contents of the CLASS_SB Descriptor

Symbol	Description
DSC$W_LENGTH DSC64$Q_LENGTH	Length of the string in bytes.
DSC64$W_MBO	Must be 1. See Section 5.1.
DSC$B_DTYPE DSC64$B_DTYPE	A data-type code that must have the value 14, which specifies the character string data type (see Sections 4.1 and 4.2). The use of other data types is reserved to Compaq.

(continued on next page)

Table 5–14 (Cont.) Contents of the CLASS_SB Descriptor

Symbol	Description
DSC$B_CLASS DSC64$B_CLASS	Defines the descriptor class code that must be equal to 15 for CLASS_SB.
DSC$A_POINTER DSC64$PQ_POINTER	Address of the first byte of data storage.
DSC64$L_MBMO	Must be -1. See Section 5.1.
DSC$L_SB_L1 DSC64$Q_SB_L1	Lower bound (signed) of the first (and only) dimension.
DSC$L_SB_U1 DSC64$Q_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):

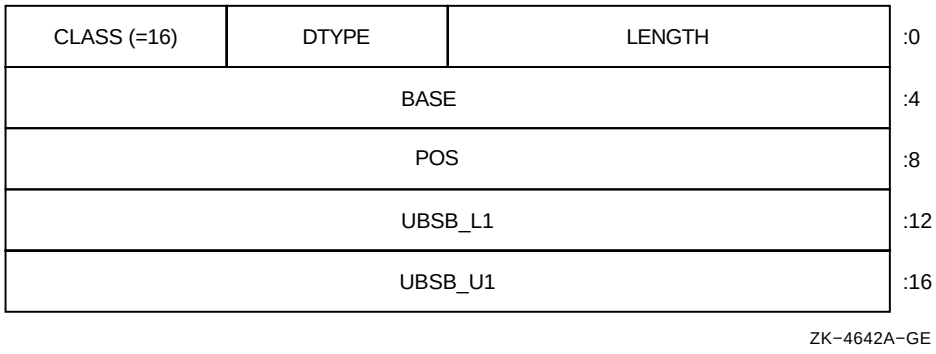
```
E = POINTER + [I - SB_L1]
```

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.

5.13 Unaligned Bit String with Bounds Descriptor (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 5–14 shows the format of an unaligned bit string with bounds descriptor. Table 5–15 describes the fields of the descriptor.

Figure 5–14 Unaligned Bit String with Bounds Descriptor Format



(continued on next page)

Figure 5–14 (Cont.) Unaligned Bit String with Bounds Descriptor Format

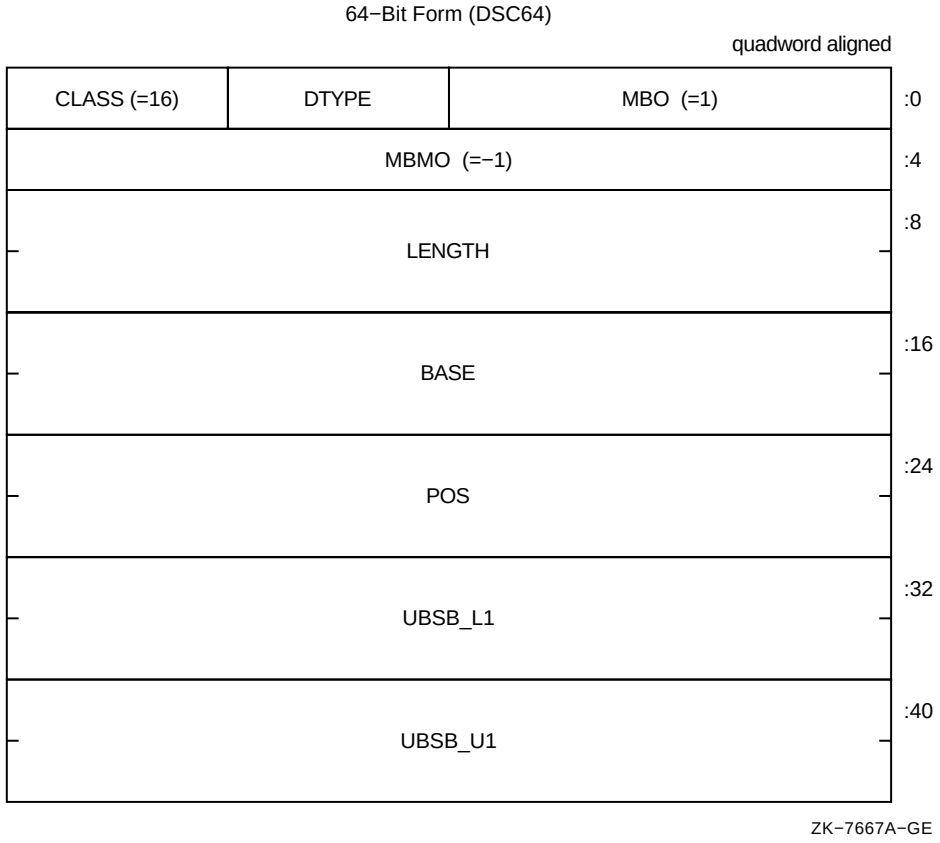


Table 5–15 Contents of the CLASS_UBSB Descriptor

Symbol	Description
DSC$W_LENGTH DSC64$Q_LENGTH	Length of the data item in bits.
DSC64$W_MBO	Must be 1. See Section 5.1.
DSC$B_DTYPE DSC64$B_DTYPE	A data-type code that must have the value 34, which specifies the unaligned bit string data type (see Sections 4.1 and 4.2). The use of other data types is reserved to Compaq.
DSC$B_CLASS DSC64$B_CLASS	Defines the descriptor class code that must be equal to 16 for CLASS_UBSB.
DSC$A_BASE DSC64$PQ_BASE	Base address relative to the signed relative bit position, POS, used to locate the bit string. The base address need not be the first actual byte of data storage.
DSC64$L_MBMO	Must be -1. See Section 5.1.
DSC$L_POS DSC64$Q_POS	Signed longword that defines the relative bit position of the first bit of the unaligned bit string to the BASE address.

(continued on next page)

Table 5–15 (Cont.) Contents of the CLASS_UBSB Descriptor

Symbol	Description
DSC$L_UBSB_L1 DSC64$Q_UBSB_L1	Lower bound (signed) of the first (and only) dimension.
DSC$L_UBSB_U1 DSC64$Q_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]
```

5.14 Reserved Descriptor Class Codes

All descriptor class codes from 0 through 191 not otherwise defined in this standard are reserved to Compaq. Classes 192 through 255 are reserved to Compaq's Computer Special Systems Group and customers.

Table 5–16 lists some specific descriptor classes and codes that are obsolete or reserved to Compaq.

Table 5–16 Specific OpenVMS VAX Descriptors Reserved to Compaq

Descriptor	Code	Class
DSC$K_CLASS_V	3	Obsolete (variable buffer)
DSC$K_CLASS_PI	6	Obsolete (procedure incarnation)
DSC$K_CLASS_J	7	Reserved to DEBUG (label)
DSC$K_CLASS_JI	8	Obsolete (label incarnation)
DSC$K_CLASS_CT	17	Reserved to ACMS (compressed text)
DSC$K_CLASS_BFA	191	Reserved to BASIC (file array)

5.14.1 Facility-Specific Descriptor Class Codes

Descriptor class codes 160 through 191 are reserved to Compaq 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 the OpenVMS Debugger.

6
OpenVMS Conditions

An OpenVMS condition is a hardware-generated synchronous exception or a software event that is to be processed in a manner similar to a VAX or an Alpha 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, and 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 exercise any of 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 sufficient.

- Take some special action and 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 6.1). 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.

6.1 Condition Values

Condition values are used in the OpenVMS operating system to provide the following functions:

- Indicate the success or failure of a called procedure as a function value
- Describe an exception condition when an exception is signaled
- Identify system messages

- Report program success or failure to the command language level

A **condition value** is a longword that includes fields to describe the software component that generates the value, the reason the value was generated, and severity status of the condition value. Figure 6–1 shows the format of a condition value. Table 6–1 describes the fields of a condition value.

Figure 6–1 Format of a Condition Value

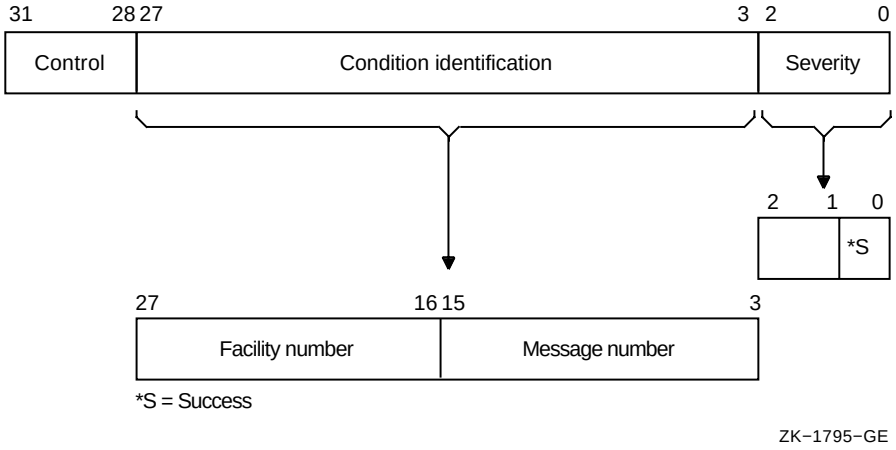


Table 6–1 Contents of the Condition Value

Symbol	Description
Severity	Indicates success or failure. The severity code bit <0> is set for success (logical true) and is 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 to Compaq
	6	Reserved to Compaq
	7	Reserved to Compaq

Section 6.1.1 more fully describes severity codes.

Condition identification	Identifies the condition uniquely on a systemwide basis.
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.

(continued on next page)

Table 6–1 (Cont.) Contents of the Condition Value

Symbol	Description
Facility number	Identifies the software component generating the condition value. Bit <27> is set for customer facilities and is clear for Compaq facilities.
Control	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 the condition has been printed or suppressed). Bits <31:29> must be 0; they are reserved to Compaq for future use. Table 6–2 lists the possible software symbols that are defined for the various fields of the condition-value longword.

Table 6–2 Value Symbols for the Condition Value Longword

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
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

(continued on next page)

Table 6–2 (Cont.) Value Symbols for the Condition Value Longword

Symbol	Value	Meaning	Field
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

6.1.1 Interpretation of Severity Codes

A standard procedure must consider all possible severity codes (0–4) of a condition value. Table 6–3 lists the interpretation of severity codes 0 through 4.

Table 6–3 Interpretation of Severity Codes

Severity Code	Meaning
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).
1	Indicates that the procedure generating the condition value completed successfully, as expected.
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.
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.
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 treats success and information as successes.
- If the condition value is signaled (see Section 6.4.3), the default handler treats severe errors as reason to terminate and treats all the others as the basis for continuation.
- When the program image exits, the command interpreter by default treats errors and severe errors as the basis for stopping the job, and treats warnings, information, and successes as the basis for continuation.

The following table summarizes the action default decisions of the severity conditions:

Severity	Routine	Signal	Default at Program Exit
Success	Normal	Continue	Continue

Severity	Routine	Signal	Default at Program Exit
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 with the SYS$OUTPUT system service. In addition, for severities other than success (STS$K_SUCCESS), a copy of the message is made on SYS$ERROR. At program exit, success and information completion values do not generate messages; however, warning, error, and severe error condition values do generate messages to SYS$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 success or severe error as its severity code for each condition value.

6.1.2 Use of Condition Values

OpenVMS software components return condition values when they complete execution. When a severity code in the range of 0 through 4 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, can 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 they do not conflict with values generated by Compaq.

6.2 Condition Handlers

To handle hardware- or software-detected exceptions, the OpenVMS Condition Handling Facility (CHF) 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. 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 that describes the condition. This condition value has the format and interpretation described in Section 6.1. All hardware exceptions are signaled.

When a condition is signaled, the Condition Handling Facility looks for a condition handler associated with the current procedure's stack frame. If a handler is found, it is entered. If a handler is not 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 until 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 that was executing when the condition was reported by the hardware. On a VAX processor, this instruction is the one immediately following the instruction that caused the underflow; on an Alpha processor, this instruction might occur later.

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.

6.3 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 6.1) that describes the condition that caused the signal. When the condition handler is entered, examine the condition value to determine the cause of the signal. After the handler either processes the condition or ignores it, it can take one of the following actions:

- Return to the instruction immediately following the signal. Note that such a return is not always possible.
- 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.
- On Alpha systems, perform a nonlocal GOTO operation (see Section 6.4) that transfers control from one procedure invocation and continues execution in a prior one.

6.4 Operations Involving Condition Handlers

The OpenVMS Condition Handling Facility (CHF) provides functions to perform the following operations:

- Establish a condition handler.

  A condition handler is associated with a procedure in various ways, depending on the language in which the procedure is written. Some languages provide specific syntax for defining a handler and its possible actions; others allow dynamic specification of a routine to act as a handler.

- On VAX systems, revert to the caller's handling.

  If a condition handler has been established on a VAX processor, you can remove it.

- 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.

  On VAX systems, exceptions are enabled or disabled dynamically at every procedure entry or by directly manipulating the processor status longword.

  On Alpha systems, exceptions are enabled or disabled statically during compilation; this is reflected in the code that is compiled.

- Signal a condition.

  Signaling a condition initiates the search for an established condition handler.

- Unwind the stack.

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

- On Alpha systems, perform a nonlocal **GOTO unwind**.

  A GOTO unwind operation is a transfer of control that leaves one procedure invocation and continues execution in a prior (currently active) procedure. This unified GOTO operation gives unterminated procedure invocations the opportunity to clean up in an orderly way.

6.4.1 Establishing a Condition Handler

On VAX systems, the association of a handler with a procedure invocation is dynamic and can be changed or reverted to the caller's handler during execution, but this is not supported for languages that implicitly provide their own handlers.

Each procedure activation can have an associated condition handler, using the first longword in its stack frame. Initially, the first longword (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.

On VAX systems, the following code establishes a condition handler:

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

On Alpha systems, the association of a handler with a procedure is static and must be specified at the time a procedure is compiled (or assembled). However, some languages that lack their own exception-handling syntax can support emulation of dynamically specified handlers by means of built-in routines.

Each procedure, other than a null frame procedure, can have a condition handler potentially associated with it, which is identified by the HANDLER_VALID, STACK_HANDLER, or REG_HANDLER fields of the associated procedure descriptor. You establish a handler by including the procedure value of the handler procedure in that field. (See Sections 3.4.1 and 3.4.4.)

In addition, the OpenVMS operating system on VAX and Alpha processors 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. For example,

the vectors are used 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, do not use them with modular code. Instead, use frame-based handlers.

6.4.2 Reverting to the Caller's Handling

On VAX systems, 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.

On VAX systems, the code to revert to the caller's handling is as follows:

```
CLRL 0(FP)
```

On Alpha systems, there is no means to revert to a caller's handler (unless a language provides emulation of dynamically specified handlers).

6.4.3 Signaling a Condition

The signal operation is the method for indicating the occurrence of an exception condition. To initiate a signal and allow execution to continue after handling the condition, a program calls the LIB$SIGNAL procedure. To initiate a signal but not allow execution to continue at the point of initiation, a program calls the LIB$STOP procedure. The format of the LIB$SIGNAL and LIB$STOP calls are defined as follows:

LIB$SIGNAL(condition-value, argn...)

LIB$STOP(condition-value, argn...)

Argument	OpenVMS Usage	Type	Access	Mechanism
condition-value	condition	longword	read	by value
argn	integer	quadword	read	by value

Arguments:

condition-value
An OpenVMS condition value.

argn
Zero or more integer arguments that become the additional arguments of a signal argument vector (see Section 6.5.1.1)

Function Value Returned:

None.

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.

Unlike most calls, LIB$SIGNAL and LIB$STOP preserve all registers. Therefore, a debugger can insert a call to LIB$SIGNAL to display the entire state of the process at the time of the exception. Use of LIB$ routines also allows signals to be coded in an assembler language without changing the register usage. This feature of preserving all registers is useful for debugging checks and for gathering statistics. Hardware and system service exceptions behave like calls to LIB$SIGNAL.

6.4.4 Signaling a Condition Using GENTRAP (Alpha Only)

On Alpha systems, a GENTRAP PALcode instruction provides an efficient means for software to raise hardwarelike exceptions. This mechanism is used in low levels of the operating system or in the bootstrap sequence when only a limited execution environment is available. In a constrained environment, GENTRAP can be handled directly using the SCB vector by which the trap is reported. In a more complete environment, the GENTRAP parameter is transformed into a corresponding exception code and is reported as a normal hardware exception. Because of this, low-level software can use this mechanism to report exceptions that are independent of the execution environment. Compiled code can also use the GENTRAP instruction to raise common generic exceptions more simply than by executing a complete LIB$SIGNAL procedure call.

The PALcode operation is defined as follows:

GENTRAP (expt_code)

The **expt_code** argument defines the code for the exception to be raised.

If the **expt_code** value is one of the small negative values shown in Table 6–4, then that value is mapped to a corresponding OpenVMS exception code as shown. If the value is negative but not one of the values shown in Table 6–4, then SS$_ GENTRAP is raised with the unmapped value included in the exception record as the first and only qualifier value. Otherwise, a positive value is used directly to raise an exception using that value as the condition value. Note that there is no means to associate any parameters with an exception raised using GENTRAP. For more information on GENTRAP, see the *Alpha Architecture Reference Manual*.

Table 6–4 Exception Codes and Symbols for the Alpha GENTRAP Argument

Exception Code	Symbol	Meaning
–1	SS$_INTOVF	Integer overflow
–2	SS$_INTDIV	Integer divide by zero
–3	SS$_FLTOVF	Floating overflow
–4	SS$_FLTDIV	Floating divide by zero
–5	SS$_FLTUND	Floating underflow
–6	SS$_FLTINV	Floating invalid operand
–7	SS$_FLTINE	Floating inexact result
–8	SS$_DECOVF	Decimal overflow
–9	SS$_DECDIV	Decimal divide by zero
–10	SS$_DECINV	Decimal invalid operand
–11	SS$_ROPRAND	Reserved operand
–12	SS$_ASSERTERR	Assertion error
–13	SS$_NULPTRERR	Null pointer error
–14	SS$_STKOVF	Stack overflow
–15	SS$_STRLENERR	String length error
–16	SS$_SUBSTRERR	Substring error

(continued on next page)

Table 6–4 (Cont.) Exception Codes and Symbols for the Alpha GENTRAP Argument

Exception Code	Symbol	Meaning
–17	SS$_RANGEERR	Range error
–18	SS$_SUBRNG	Subscript range error
–19	SS$_SUBRNG1	Subscript 1 range error
–20	SS$_SUBRNG2	Subscript 2 range error
–21	SS$_SUBRNG3	Subscript 3 range error
–22	SS$_SUBRNG4	Subscript 4 range error
–23	SS$_SUBRNG5	Subscript 5 range error
–24	SS$_SUBRNG6	Subscript 6 range error
–25	SS$_SUBRNG7	Subscript 7 range error

6.4.5 Condition Handler Search

The signal procedure examines the two exception vectors first, then examines a system-defined maximum number of previous stack frames, and, if necessary, examines the last-chance exception vector. The current and previous stack frames are found by using the frame pointer and chaining back through the stack frames using the saved context 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.

On Alpha systems, note that the default catchall handler in user mode can be a list of handlers and is not in conflict with this standard.

On both VAX and Alpha systems, in some frame before the call to the main program, the system establishes a default catchall 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 SYS$PUTMSG system service.

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 frame address is 0 or is not accessible, or when a system-defined maximum number of frames have been examined. If a condition handler is not found, or if all handlers return with a SS$_RESIGNAL or SS$_RESIGNAL64, then the vectored last-chance handler is called.

If a handler returns SS$_CONTINUE or SS$_CONTINUE64, 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 6–2 lists all combinations of interaction between condition handler actions, default condition handlers, types of signals, and calls to signal or stop. In this figure, "Cannot Continue" indicates an error that results in the following message:

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

Figure 6–2 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

6.5 Properties of Condition Handlers

This section describes the properties of condition handlers for both VAX and Alpha environments.

6.5.1 Condition Handler Parameters and Invocation

If a condition handler is found on a software-detected exception, the handler is called as follows:

handler(signal_args, mechanism_args)

Argument	OpenVMS Usage	Type	Access	Mechanism
signal_args	signal vector	structure	modify	by reference
mechanism_args	mechanism	structure	modify	by reference

Arguments:

signal_args
A 32-bit signal argument vector (see Section 6.5.1.1)

mechanism_args
A mechanism argument vector (see Section 6.5.1.2)

Function Value Returned:

One of the following status codes: SS$_CONTINUE, SS$_RESIGNAL, SS$_CONTINUE64, SS$_RESIGNAL64. This value is used by the Condition Handling Facility to determine how to proceed next in processing the condition. (See Section 6.6.)

6.5.1.1 Signal Argument Vector

There are two forms of signal argument vector (or signal vector for short): one for use with 32-bit addresses and one for use with 64-bit addresses. The two forms are compatible in that the forms can be distinguished dynamically at run time and, except for the size and offset of fields, are identical in content and interpretation.

The 32-bit signal argument vectors are used on both OpenVMS VAX and OpenVMS Alpha systems. When used on OpenVMS Alpha, 32-bit signal argument vectors provide full compatibility with their use on OpenVMS VAX. The 64-bit signal argument vectors are used only on OpenVMS Alpha—they have no counterpart and are not recognized on OpenVMS VAX systems.

When a condition handler is called by the Condition Handling Facility (CHF) on Alpha, both forms of signal argument vector are available. The first argument is always a reference to a 32-bit form of signal argument vector. A handler that chooses to operate using the 64-bit form must obtain the address of the corresponding 64-bit signal argument vector from the CHF$PH_MCH_SIG64_ADDR field of the mechanism argument vector (see Section 6.5.1.2).

Both forms of signal vector include a length field, a condition value, zero or more parameters that further qualify the condition value, and finally a processor program counter (PC) and program status (PS). For hardware-detected exceptions, the condition value indicates which exception was taken. The PC value gives the address of the instruction that caused the exception or the address of the next instruction, depending on whether the exception was a fault or a trap. For software-detected conditions, the condition value and any associated parameters are copies of the parameters to the call of LIB$SIGNAL or LIB$STOP that initiated exception handling, while the PC is the return address to the caller of that routine.

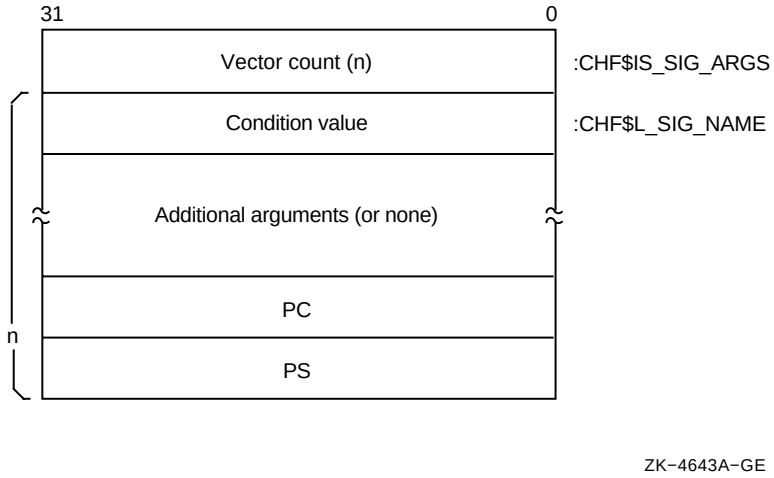
Note that bits <2:0> of a condition value indicate severity and not what condition is being signaled. Therefore, a handler should examine only the condition identification, that is, condition value bits <27:3>, to determine the cause of the exception. The setting of severity bits <2:0> may vary from time to time even for the same condition. In fact, some handlers might only change the severity of a condition in the signal vector and resignal.

Generally, a handler may validly modify any field of a signal argument vector except for the CHF$L_SIG_ARGS length field or, in the case of a 64-bit signal vector, the CHF64$L_SIGNAL64 field. In particular, a modified signal vector is passed to a subsequent handler if the current handler completes by resignaling. (If the length is modified, the modification is ignored; CHF restores the original length.) It is invalid for a handler to modify both forms of signal argument vector—the effect of doing so is undefined.

The remainder of this section is organized as follows. First, the 32-bit form of signal argument vector is described. Second, the 64-bit form of signal argument is described. Finally, the relationship between the two forms is discussed.

Figure 6–3 shows the format of the 32-bit form of signal argument vector. The CHF$L_SIG_ARGS longword contains the argument vector count, which is the number of remaining longwords in the vector. The CHF$L_SIG_NAME longword contains the condition value. Next are 0 or more longwords that contain additional parameters appropriate to the condition. The remaining two longwords contain the PC and PS values.

Figure 6–3 Signal Argument Vector — 32-Bit Format

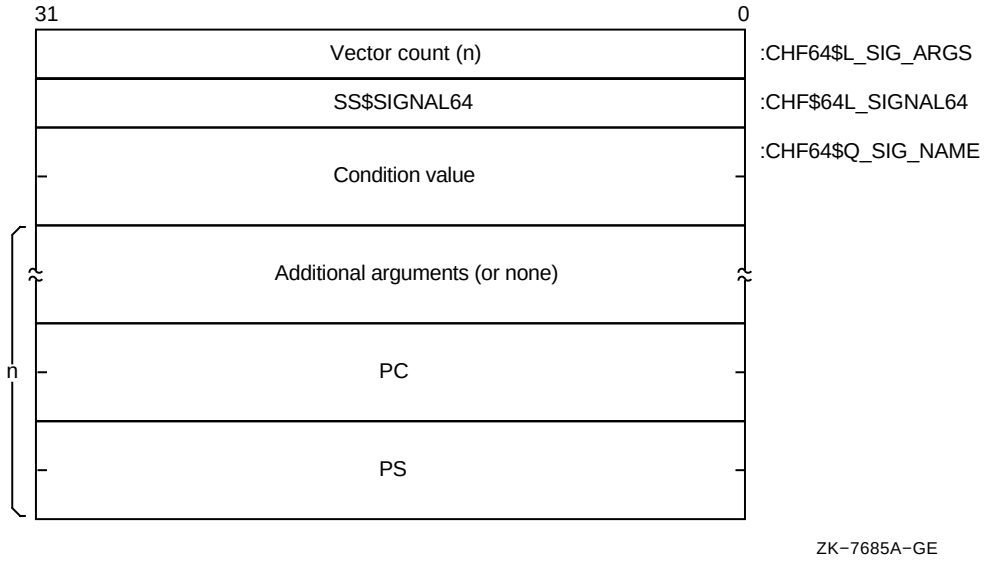


On VAX systems, the value used for the PS is the contents of the VAX processor status longword (PSL).

On Alpha systems, the value used for the PS is the low half of the Alpha processor status register. Furthermore, CHF$IS_SIG_ARGS and CHF$IS_SIG_NAME are aliases for CHF$L_SIG_ARGS and CHF$L_SIG_NAME, respectively.

Figure 6–4 shows the format of the 64-bit form of signal argument vector. The address of this form of signal argument is available only from the CHF$PH_MCH_SIG64_ADDR field of the mechanism argument vector (see Section 6.5.1.2). The CHF64$L_SIG_ARGS field is a longword that contains the number of remaining quadwords in the vector (following the CHF64$L_SIGNAL64 field). The CHF64$L_SIGNAL64 longword contains a special code named SS$_SIGNAL64 whose value is key to distinguishing between a 32-bit and 64-bit form of signal argument vector. The CHF64$Q_SIG_NAME quadword contains a sign-extended condition value. Next are zero or more quadwords that contain additional parameters appropriate to the condition. The remaining two quadwords contain the Alpha PC and PS values.

Figure 6–4 Signal Argument Vector — 64-Bit Format



When a handler is called, the 32-bit and 64-bit signal argument vectors are closely related as follows:

- The value of the length field in the 64-bit form (the number of quadwords following the CHF64$L_SIGNAL64 field) is equal to the value of the length field in the 32-bit form (the number of longwords following the CHF$L_SIG_ ARGS field).
- The condition value, any related arguments, and the PC and PS values in the 32-bit form are the same as the values in the 64-bit form truncated to 32 bits.

Note that given a 64-bit signal vector, it is possible to create the corresponding 32-bit signal vector by fetching the low-order longword of each quadword of the 64-bit vector and packing the results together contiguously into a 32-bit vector; other than using the length, no interpretation of the contents is required.

Given the address of a signal argument vector that might be either the 32-bit or 64-bit form, either of the following equivalent tests may be used to distinguish which one is present:

- Assuming a 32-bit form, compare the contents of the CHF$L_SIG_NAME field (equivalently CHF64$L_SIGNAL64) with the value SS$_SIGNAL64. If equal, then the 64-bit form is present; otherwise, the 32-bit form is present.
- Assuming a 64-bit form, compare the contents of the CHF64$L_SIGNAL64 field with the value SS$_SIGNAL64. If equal, then the 64-bit form is present; otherwise, the 32-bit form is present.

6.5.1.2 Mechanism Argument Vector

The mechanism argument vector for the argument **mechanism_args** contains information about the machine state when an exception occurs or when a condition is signaled. Therefore, the mechanism argument vector is highly specific to the underlying machine architecture.

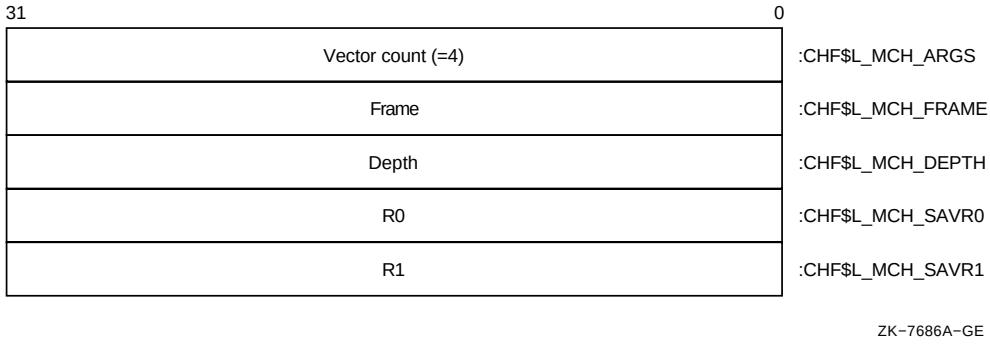
On VAX systems, the mechanism format for the argument vectors is shown in Figure 6–5. The first longword contains the argument vector count, which is the number of remaining longwords in the vector. The frame longword contains the contents of the FP in the establisher's context. If the restrictions described in Section 6.5.3.1 are met, the frame can be used as a base from which to access the local storage of the establisher.

The depth longword 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. (For more information about depth, see Section 6.5.1.3.)

The CHF$L_MCH_SAVR0 and CHF$L_MCH_SAVR1 longwords save the state of the R0 and R1 registers, respectively, at the time of the call to LIB$SIGNAL or LIB$STOP. If not modified by a handler during CHF processing, these values will become the values of those registers after completion of CHF processing (either by continuation or by unwinding). These two fields may be modified by a handler to establish different values to be used at CHF completion. Note that the contents of other registers are not available in the mechanism vector and can only be accessed by analysis of the stack. (See Section 6.7.1.)

CHF$L_MCH_SAVR0 and CHF$L_MCH_SAVR1 are the only fields of a VAX mechanism vector that can be validly modified by a handler. The effect of any other modification is undefined.

Figure 6–5 VAX Mechanism Vector Format



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. (Save and restore is not required for exceptions 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 2.1.2.) Thus, a condition handler can make use of the system vector facilities in the same manner as mainline code.

The VAX 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.)

On Alpha systems, the 64-bit-wide mechanism array is the argument mechanism in the handler call. The array shown in Figure 6–6 is defined by constant CHF$S_CHFDEF2 at a size of 360 bytes (45 quadwords). Table 6–5 lists and describes the fields.

The CHF$IH_MCH_SAVR*nn* and CHF$FH_MCH_SAVF*nn* quadwords save the state of the nonpreserved general and floating registers, respectively, at the time of the call to LIB$SIGNAL or LIB$STOP. If not modified by a handler during CHF processing, these values will become the values of those registers after completion of CHF processing (either by continuation or by unwinding). These fields may be modified by a handler to establish different values to be used at CHF completion.

The CHF$IH_MCH_SAVR*nn* and CHF$FH_MCH_SAVF*nn* fields are the only fields of an Alpha mechanism vector that can be validly modified by a handler. The effect of any other modification is undefined. (See also Section 6.7.2.) Note that the contents of the normally preserved registers are not available in the mechanism vector and can only be accessed by analysis of the stack. (See Section 6.7.1.)

Figure 6–6 Alpha Mechanism Vector Format

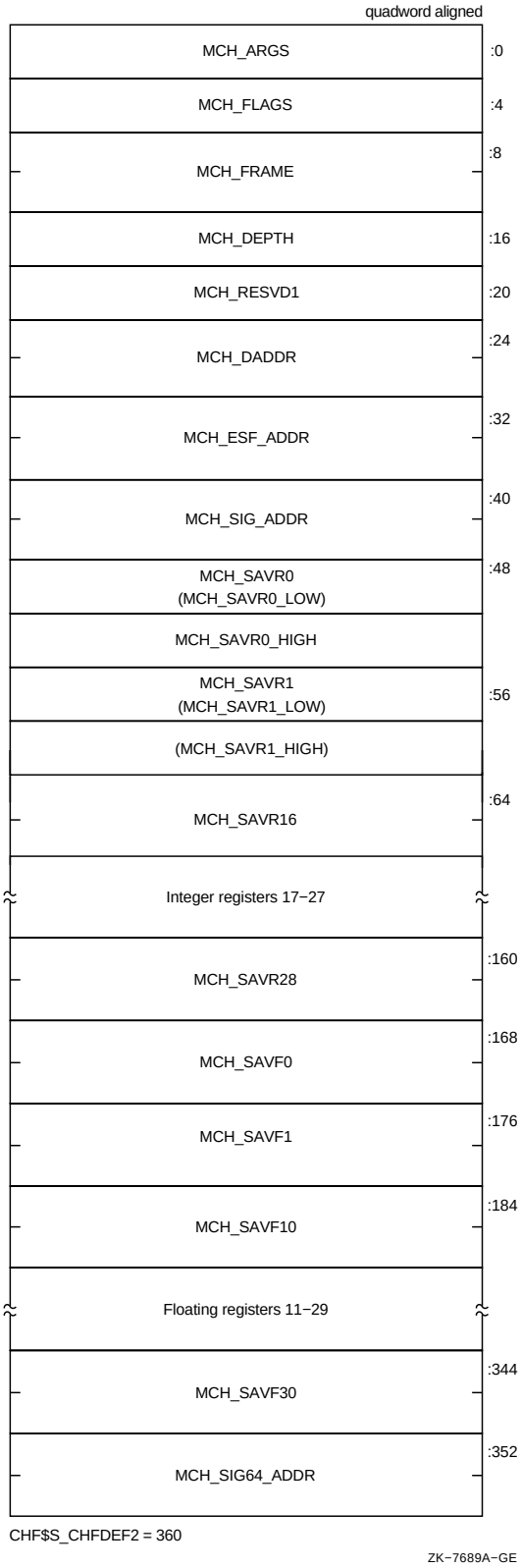


Table 6–5 Contents of the Alpha Argument Mechanism Array (MECH)

Field Name	Contents
CHF$IS_MCH_ARGS	Count of quadwords in this array starting from the next quadword, CHF$IS_MCH_FRAME (not counting the first quadword that contains this longword). This value is always 44.
CHF$IS_MCH_FLAGS	Flag bits <31:0> for related argument-mechanism information defined as follows: CHF$V_FPREGS_VALID — Bit 0. When set, the process has already performed a floating-point operation and the floating-point registers stored in this structure are valid. If this bit is clear, the process has not yet performed any floating-point operations and the values in the floating-point register slots in this structure are unpredictable.
CHF$PH_MCH_FRAME	Contains the frame pointer in the procedure context of the establisher.
CHF$IS_MCH_DEPTH	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 (see Section 6.5.1.3).
CHF$IS_MCH_RESVD1	Reserved to Compaq.
CHF$PH_MCH_DADDR	Address of the handler data quadword if the exception handler data field is present (as indicated by PDSC$V_HANDLER_DATA_VALID); otherwise, contains 0.
CHF$PH_MCH_ESF_ADDR	Address of the exception stack frame (see the *Alpha Architecture Reference Manual*).
CHF$PH_MCH_SIG_ADDR	Address of the 32-bit form of signal array. This array is a 32-bit wide (longword) array. This is the same array that is passed to a handler as the signal argument vector.
CHF$PH_MCH_SIG64_ ADDR	Address of the 64-bit form of signal array. This array is a 64-bit wide (quadword) array.
CHF$IH_MCH_SAVR*nn*	Contain copies of the saved integer registers at the time of the exception. The following registers are saved: R0, R1, and R16 through R28. Registers R2 through R15 are implicitly saved in the call chain.
CHF$FH_MCH_SAVF*nn*	Contain copies of the saved floating-point registers at the time of the exception, or are unpredictable as described at field CHF$IS_MCH_ FLAGS. If the floating-point register fields are valid, the following registers are saved: F0, F1, and F10 through F30. Registers F2 through F9 are implicitly saved in the call chain.

6.5.1.3 Mechanism Depth for Alpha and VAX Handler Arguments

For Alpha and VAX argument mechanisms, the depth field has the value 0 for an exception that is handled by the procedure activation invoking the exception. The exception procedure contains the instruction that either causes the hardware exception or calls LIB$SIGNAL. The depth field of the argument mechanism 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. The depth field has a value of −2 when the condition handler is established by the primary exception vector, a value of −1 when it is established by the secondary vector, and a value of −3 when it is established by the last-chance vector.

The Alpha mechanism depth may not be the same as the depth for the same circumstances on a VAX system if any of the following are present:

- Condition dispatcher in the call chain
- Jacket frames, if there are any translated routines in the call chain
- Multiple active signals
- Compiler use of no frame procedures or inline code expansion of calls

6.5.2 System Default Condition Handlers

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

6.5.3 Coordinating the Handler and Establisher

This section describes the requirements for use of memory, exception synchronization, and continuation of the handler.

6.5.3.1 Use of Memory

Exceptions can be raised and unwind operations (which cause exception handlers to be called) can occur when the current value of one or more variables is in registers rather than in memory. Because of this, a handler, and any descendant procedure called directly or indirectly by a handler, must not access any variables except those explicitly passed to the procedure as arguments or those that exist in the normal scope of the procedure.

This rule can be violated for specific memory locations only by agreement between the handler and all procedures that might access those memory locations. A handler that makes such agreements does not conform to this standard.

6.5.3.2 Exception Synchronization (Alpha Only)

The Alpha hardware architecture allows instructions to complete in a different order than that in which they were issued, and for exceptions caused by an instruction to be raised after subsequently issued instructions have been completed.

Because of this, the state of the machine when a hardware exception occurs cannot be assumed with the same precision as it can be assumed on conventional VAX machines unless such precision has been guaranteed by bounding the exception range with the appropriate insertion of TRAPB instructions.

The rules for bounding the exception range follow:

- If a procedure has an exception handler that does not simply reraise all arithmetic traps caused by code that is not contained directly within that procedure, the procedure must issue a TRAPB instruction before it establishes itself as the current procedure.

- If a procedure has an exception handler that does not simply reraise all arithmetic traps caused either by code that is not contained directly within that procedure or by any procedure that might have been called while that procedure was current, the procedure must issue a TRAPB instruction in the procedure epilogue while it is still the current procedure.
- If a procedure has an exception handler that is sensitive to the invocation depth, the procedure must issue a TRAPB instruction immediately before and after any call. Furthermore, the handler must be able to recognize exception PC values that represent either epilogue code in called procedures or TRAPB instructions immediately after a call, and adjust the depth appropriately (see Section 3.7.5).

These rules ensure that exceptions are detected in the intended context of the exception handler.

These rules do *not* ensure that all exceptions are detected while the procedure within which the exception-causing instruction was issued is current. For example, if a procedure without an exception handler is called by a procedure that has an exception handler not sensitive to invocation depth, an exception detected while that called procedure is current may have been caused by an instruction issued while the caller was the current procedure. This means the frame, designated by the exception-handling information, is the frame that was current when the exception was detected, not necessarily the frame that was current when the exception-causing instruction was issued.

6.5.3.3 Continuation from Exceptions (Alpha Only)

The Alpha architecture guarantees neither that instructions are completed in the same order in which they were fetched from memory nor that instruction execution is strictly sequential. Continuation is possible after some exceptions, but certain restrictions apply.

By definition, software-raised general exceptions are synchronous with the instruction stream and can have a well-defined continuation point. Therefore, a handler can request continuation from a software-raised exception. However, since compiler-generated code typically relies on error-free execution of previously executed code, continuing from a software-raised exception might produce unpredictable results and unreliable behavior unless the handler has explicitly fixed the cause of the exception so that it is transparent to subsequent code.

Hardware faults on Alpha processors follow the same rules as the strict interpretation of the conventional VAX rules. Loosely paraphrased, these rules state that if the offending exception is fixed, reexecution of the instruction (as determined from the supplied PC) will yield correct results. This does *not* imply that instructions following the faulting instruction have not been executed. Therefore, hardware faults can be viewed as similar to software-raised exceptions and can have well-defined continuation points.

Arithmetic traps cannot be restarted because all the information required for a restart is not available. The most straightforward and reliable way in which software can guarantee the ability to continue from this type of exception is by placing appropriate TRAPB instructions in the code stream. Although this technique does allow continuation, it must be used with extreme caution because of the negative effect on application performance.

6.6 Returning from a Condition Handler

Condition handlers are invoked by the OpenVMS Condition Handling Facility (CHF). Therefore, the return from the condition handler is to the CHF.

To continue from the instruction following the signal, the handler must return with a function value of either SS$_CONTINUE or SS$_CONTINUE64 (both of which have bit <0> set). If, however, the condition is signaled with a call to LIB$STOP, the image exits. To resignal the condition, the condition handler returns with a function value of either SS$_RESIGNAL or SS$_RESIGNAL64 (both of which have the bit <0> clear).

The difference between SS$_CONTINUE and SS$_CONTINUE64, and similarly between SS$_RESIGNAL and SS$_RESIGNAL64, is of significance only if the handler has made an alteration to the signal vector that is intended to be taken into account by the CHF. When SS$_CONTINUE or SS$_RESIGNAL is returned, then any modification to the 32-bit signal vector is propagated (in sign-extended form) to the corresponding position in the 64-bit vector. When SS$_CONTINUE64 or SS$_RESIGNAL64 is returned, any modification in the 64-bit signal vector is propagated (in truncated form) to the corresponding position in the 32-bit vector. If no modification has been made, then the two forms of continuation or resignal are equivalent.

The algorithm for detecting change is as follows:

- For SS$_CONTINUE64 and SS$_RESIGNAL64, the 32-bit signal vector is simply derived again from the 64-bit signal vector. In particular, no hidden copy of the 64-bit signal vector is kept. It is not necessary to determine if there was a change or not—if there was, it is properly reflected in the 32-bit vector.
- For SS$_CONTINUE and SS$_RESIGNAL, let SIGVEC32[I] and SIGVEC64[I] be corresponding entries in the two vectors, for I from 1 to length. (Recall that the length[s] cannot be changed.) For each entry, do the following:

  ```
  if SIGVEC32[I] /= SIGVEC64[I]<0,32>
  then
      SIGVEC64[I] = sign-extend(SIGVEC32[I])
  ```

  That is, if the 32-bit entry is still the same as the low-order 32 bits of the 64-bit entry, then it did not change and thus the 64-bit entry is not changed. Otherwise, update the 64-bit entry with the sign-extended contents of the 32-bit entry.

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.

6.7 Request to Unwind from a Signal

To unwind, the handler or any procedure that it calls can make a call to SYS$UNWIND. The format is as follows:

SYS$UNWIND(depadr, new_PC)

Argument	OpenVMS Usage	Type	Access	Mechanism
depadr	integer	longword	read	by reference
new_PC	address	longword	read	by reference

Arguments:

depadr
Optional number of presignal frames (depth) to be removed.

new_PC
Optional address of the location to receive control after the unwind operation is completed.

Function Value Returned:

Success or failure status (see text that follows).

The **depadr** argument specifies the address of the longword that contains the number of presignal frames (depth) to be removed. The deepest procedure invocation whose frame is not removed is called the **target invocation** of the unwind. If that number is less than or equal to 0, 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 $UNWIND service. To unwind to the establisher, specify the depth from the call to the handler, which can be found in the CHF$IS_MCH_DEPTH field of the Mechanism Array. 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$_CONTINUE, or SS$_CONTINUE64 if the 64-bit signal vector is altered, instead of performing an unwind.

The **new_PC** argument 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 that is removed from the stack.

The function value **success** either is 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)

If SYS$UNWIND is invoked by a handler that has already invoked SYS$UNWIND, then the effect of the second invocation is undefined.

The unwinding operation occurs when the handler returns to the CHF. Unwinding is done by scanning back through the stack and calling each handler associated with a frame. The handler is called with the 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.

When the target invocation is reached on Alpha systems, unwind completion depends on the PDSC$V_TARGET_INVO flag of the associated procedure descriptor. If that flag is set to 1, then the handler for that procedure invocation is called; otherwise, no handler is called. Control then resumes in the target invocation.

The call to the handler takes the same form as described in Section 6.5.1 with the following values:

- **signal_args**: for a handler for a procedure other than the target invocation of the unwind—an argument count (CHF$L_SIG_ARGS) of 1 and a condition value (CHF$L_SIG_NAME) of SS$_UNWIND.

  For a handler on Alpha systems for a procedure that is the target invocation of the unwind—an argument count (CHF$L_SIG_ARGS) of 2 and two condition values consisting of SS$_UNWIND followed by SS$_TARGET_ UNWIND.

- **mechanism_args**: same as for the original call except for a depth of 0 (that is, unwinding self) and any other changes made by prior handlers.

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

On Alpha systems, the stack is not actually cut back until after the last handler is called.

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 the appropriate saved register locations 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$UNWIND(0,0)	Unwind to the establisher's caller.
SYS$UNWIND(*depth*,0)	Unwind to the establisher at the point of the call that resulted in the exception.
SYS$UNWIND(*depth*,*location*)	Unwind to the specified procedure activation and transfer to a specified location.

The only recommended values for *depth* are the default (address of 0), which unwinds to the caller of the establisher, and the value of *depth* taken from the mechanism vector, which unwinds to the establisher. Other values depend on implementation details that can change at any time.

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.

6.7.1 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.

On VAX systems, to find 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. In this case, the registers are in the following states:

R0, R1	In **mechanism_args**
R2–11	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

On Alpha systems, to find the contents of the registers, use the invocation context routines described in Section 3.6.3.

6.7.2 Unwind Completion

On VAX systems, the values that exist in R0 and R1 when the unwind completes are the values passed implicitly to the unwinder in the mechanism array (see Section 6.5.1.2). If desired, these values can be modified by an exception handler before the unwind is initiated.

On Alpha systems, the values that exist in R0, R1, F0, and F1 when the unwind completes are the values passed implicitly to the unwinder in the mechanism array (see Section 6.5.1.2). If desired, these values can be modified by an exception handler before the unwind is initiated. Note that, unlike VAX systems, an Alpha system does not use R1 for returning any type of return values.

6.8 GOTO Unwind Operations (Alpha Only)

A **GOTO unwind** is a transfer of control that leaves one procedure invocation and continues execution in a prior, currently active procedure invocation. Modular and reliable support of the nonlocal GOTO requires procedure invocations that are terminated to have an opportunity to clean up in an orderly way (just like a procedure that is terminated as a result of an unwind from a condition handler).

Performing a GOTO unwind operation in a thread causes a transfer of control from the location at which the GOTO unwind operation is initiated to a target location in a target invocation. This transfer of control also results in the termination of all procedure invocations, including the invocation in which the unwind request was initiated, up to the target procedure invocation. Thread execution then continues at the target location.

Before control is transferred to the unwind target location, the unwind support code invokes all frame-based handlers that were established by procedure invocations being terminated. These handlers are invoked with an indication of an unwind in progress. This gives each procedure invocation being terminated the chance to perform cleanup processing before its context is lost.

When the target invocation is reached, unwind completion depends on the PDSC$V_TARGET_INVO flag of the associated procedure descriptor. If that flag is set to 1, then the handler for that procedure invocation is called; otherwise, no handler is called.

After all the relevant frame-based handlers have been called and the appropriate frames have been removed from existence, the target invocation's saved context is restored and execution is resumed at the specified location.

A GOTO unwind procedure can be initiated while an exception is active (from within a condition handler) or while no exception is active. If the GOTO unwind transfers control out of an exception handler (resulting in the termination of current handler invocation), it also terminates handling of the corresponding condition (like SYS$UNWIND).

A GOTO unwind operation in which a target invocation is not specified is called an **exit unwind**. An exit unwind is just like a GOTO unwind except that every procedure invocation in the currently executing thread is terminated. An exit unwind is the only standard way to terminate execution of the currently executing thread (other than a normal return from the topmost procedure of the thread).

A thread can initiate a GOTO unwind or an exit unwind operation by calling a system service routine. This routine is defined as follows:

SYS$GOTO_UNWIND(target_invo, target_pc, new_R0, new_R1)

Argument	OpenVMS Usage	Type	Access	Mechanism
target_invo	invo_handle	longword (unsigned)	read	by reference
target_pc	address	longword (unsigned)	read	by reference
new_R0	quadword_unsigned	quadword (unsigned)	read	by reference
new_R1	quadword_unsigned	quadword (unsigned)	read	by reference

Arguments:

target_invo
Address of a location that contains a handle for the target invocation.

If omitted or the address of the handle is 0, then an exit unwind is initiated.

target_pc
Address of a location that contains the address at which execution should continue in the target invocation.

If omitted or if the address is 0, then execution resumes at the location specified by the return address for the call frame of the target procedure invocation.

If the **target_invo** argument is omitted or is 0, then this argument is ignored. In this case, a system-defined target PC is assumed.

new_R0
Address of a location that contains the value to place in the saved R0 location of the mechanism argument vector. The contents of this location are then loaded into R0 at the time that execution continues in the target invocation.

If this argument is omitted, then the contents of the processor R0 register at the time of the call to SYS$GOTO_UNWIND are used.

new_R1
Address of a location that contains the value to place in the saved R1 location of the mechanism argument vector. The contents of this location are then loaded into R1 at the time that execution continues in the target invocation.

If this argument is omitted, then the contents of R1 at the time of the call to SYS$GOTO_UNWIND are used.

Condition Value Returned:

SS$_ACCVIO
An invalid address was given.

When a GOTO unwind is initiated, control usually never returns to the point at which the unwind was initiated. Control returns with an error status only if a GOTO unwind cannot be started. If SYS$GOTO_UNWIND is invoked by a handler that has already invoked SYS$UNWIND, then the effect of calling SYS$GOTO_UNWIND is undefined.

6.8.1 Handler Invocation During a GOTO Unwind

When an unwind operation takes place, all frame-based exception handlers are invoked that were established by any procedure invocation being terminated. In addition, the handler for the target procedure invocation is called if the PDSC$V_TARGET_INVO flag is set in the corresponding procedure descriptor (see Sections 3.4.2 and 3.4.5.) These handlers are invoked in the reverse order from which they were established.

Since primary, last-chance handlers, and the system catchall handler are not associated with a normal procedure invocation, these handlers are never invoked during an unwind (but they are invoked if an exception is raised during the unwind operation).

For a GOTO or exit unwind procedure, each handler that is invoked is called with two arguments as follows:

handler (signal_args, mechanism_args)

Argument	OpenVMS Usage	Type	Access	Mechanism
signal_args	signal vector	structure	modify	by reference
mechanism_args	mechanism vector	structure	modify	by reference

Arguments:

signal_args
Argument count of 2, followed by a condition value of SS$_UNWIND, followed by:

- SS$_EXIT_UNWIND when no target invocation is specified
- SS$_GOTO_UNWIND when a target invocation is specified but not for that target invocation
- SS$_TARGET_GOTO_UNWIND when a target invocation is specified and the handler for that target invocation is called

mechanism_args
Mechanism argument corresponding to the frame being unwound, as defined in Section 6.5.1.2.

For information about signal argument and mechanism argument vectors, see Sections 6.5.1.1 and 6.5.1.2.

6.8.2 Unwind Completion

When an unwind completes, the following conditions are true:

- The target procedure invocation is the most current invocation in the procedure invocation chain.
- The environment of the target invocation is restored to the state when that invocation was last current, except for the contents of all scratch registers.
- R0 and R1 contain the respective values, if any, which were passed by the routine that invoked the unwind.

- Execution continues at the target location.

6.9 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:

<Signal T>
Y
X
Bh
<Signal S>
C
B
A

Which was programmed:

The handlers are searched for in the following order: Yh, Xh, Bhh, Ah. Bh is not called again because it is not appropriate to assume that a routine is able to be its own handler. However, Bh can establish itself or another procedure as its handler (Bhh).

On VAX systems, Ch is not checked or called because it is a structural descendant of B.

On Alpha systems, the search does check handlers Ch and Bh between calling Bhh and Ah. These handlers will be reinvoked only if enabled by the HANDLER_REINVOCABLE flag of the establisher's procedure descriptor (see Sections 3.4.1 and 3.4.4).

For both Alpha and VAX systems, 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. However, the search backward in the process stack is then modified. On a VAX processor, the stack frames traversed in the first search are skipped, in effect, during the second search, while on an Alpha processor, the stack frames are skipped unless they explicitly enable handler reinvocation. Therefore, the stack frame preceding the first condition handler, up to and including the frame of the procedure that has established the handler, is skipped.

In the VAX environment, frames that are skipped are not counted in the depth. In the Alpha environment, all frames are counted in the depth.

For example, the stack frames traversed in the first and second searches are skipped 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, the second handler is invoked. Therefore, 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 unwind 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.

6.10 Multiple Active Unwind Operations

During an unwind operation (resulting from either SYS$GOTO_ UNWIND or SYS$UNWIND), another unwind operation can be initiated (using either SYS$GOTO_UNWIND or SYS$UNWIND). This can occur, for example, if a handler that is invoked for the original unwind initiates another unwind, or if an exception is raised in the context of such a handler and a handler invoked for that exception initiates another unwind operation. However, SYS$UNWIND cannot be called from a handler that is invoked as part of an unwind (see Section 6.7), but it can be called from a handler for a nested exception.

An unwind that is initiated while a previous unwind is active is either a nested unwind or an overlapping unwind.

A **nested unwind** is an unwind that is initiated while a previous unwind is active and whose target invocation in the procedure invocation chain is not a predecessor of the most current active unwind handler. A nested unwind does not terminate any procedure invocation that would have been terminated by the previously active unwind.

When a nested unwind is initiated, no special rules apply. The nested unwind operation proceeds as a normal unwind operation, and when execution resumes at the target location of the nested unwind, the nested unwind is complete and the previous unwind is once again the most current unwind operation.

An **overlapping unwind** is an unwind that is initiated while a previous unwind is active and whose target invocation in the procedure invocation chain is a predecessor of the most current active unwind handler. An overlapping unwind terminates one or more procedure invocations that would have been terminated by the previously active unwind.

An overlapping unwind is detected when the most current active unwind handler is terminated. This detection of an overlapping unwind is termed an **unwind collision**.

When a GOTO unwind collides with a GOTO unwind, the later unwind supersedes the earlier unwind, which is abandoned. The later unwind then continues from the point of the collision.

The result of any other collision is undefined.

Index

A

B

C

D

E

F

G

H

I

L

M

N

O

P

R

S

T

U

V