Tru64 UNIX

Calling Standard for Alpha Systems

Part Number: AA-RH9MC-TE

August 2000

Product Version: Tru64 UNIX Version 5.1 or higher

This manual defines the requirements, mechanisms, and conventions used in the Tru64 UNIX interface that supports procedure calls for Tru64 UNIX on Alpha systems.

Hewlett-Packard Company
Palo Alto, California

Contents

About This Manual

1 Introduction

1.1 Applicability 1–2
1.2 Architectural Level 1–2
1.3 Goals 1–3
1.4 Requirements 1–4
1.5 Definitions 1–4

2 Tru64 UNIX Concepts

2.1 Address Representation 2–1
2.2 Procedure Representation 2–1
2.3 Register Usage Conventions 2–2
2.3.1 Integer Registers 2–2
2.3.2 Floating-Point Registers 2–3
2.4 Register Names 2–4
2.5 Program Image Layout 2–4

3 Flow Control

3.1 Procedure Types 3–1
3.1.1 Procedure Descriptor Overview 3–1
3.1.2 Stack Frame Procedure 3–4
3.1.2.1 Stack Frame Format 3–4
3.1.2.2 Register Save Area 3–7
3.1.3 Register Frame Procedure 3–10
3.1.4 Null Frame Procedure 3–11
3.2 Transfer of Control 3–11
3.2.1 Call Conventions 3–12
3.2.2 Linkage 3–14
3.2.3 Link-Time Optimization 3–15
3.2.4 Calling Computed Addresses 3–15
3.2.5 Bound Procedure Values 3–16
3.2.6 Entry and Exit Code Sequences 3–17
3.2.6.1 Entry Code Sequence 3–17

3.2.6.1.1 Prologue Length 3–19
3.2.6.1.2 Frame Pointer Conventions 3–19
3.2.6.1.3 Entry Code Example for a Stack Frame Procedure 3–19
3.2.6.1.4 Entry Code Example for a Register Frame Procedure 3–20
3.2.6.2 Exit Code Sequence 3–20
3.2.6.2.1 Reserved Instruction Sequence for a Procedure Exit 3–21
3.2.6.2.2 Exit Code Sequence Steps 3–23
3.2.6.2.3 Exit Code Example for a Stack Frame Procedure . 3–24
3.2.6.2.4 Exit Code Example for a Register Frame Procedure 3–24

4 Data Manipulation

4.1 Data Passing 4–1
4.1.1 Argument Passing Mechanisms 4–1
4.1.2 Normal Argument List Structure 4–2
4.1.3 Homed Memory Argument List Structure 4–3
4.1.4 Argument Lists and High-Level Languages 4–5
4.1.5 Unused Bits in Passed Data 4–5
4.1.6 Sending Data 4–7
4.1.6.1 Sending Mechanism 4–8
4.1.6.2 Order of Argument Evaluation 4–9
4.1.7 Returning Data 4–9
4.1.7.1 Function Value Return by Immediate Value 4–10
4.1.7.2 Function Value Return by Reference 4–10
4.1.7.3 Function Value Return by Descriptor 4–11
4.2 Data Allocation 4–11
4.2.1 Data Alignment 4–12
4.2.2 Granularity of Memory 4–13
4.2.3 Record Layout Conventions 4–13

5 Event Processing

5.1 Exception Handling 5–1
5.1.1 Exception Handling Requirements 5–2
5.1.2 Exception Handling Overview 5–2
5.1.3 Kinds of Exceptions 5–3
5.1.3.1 General Exceptions 5–4
5.1.3.2 Unwind Exceptions 5–4
5.1.3.3 Signal Exceptions 5–5

5.1.4 Status Values and Exception Codes 5–5
5.1.5 Exception Records 5–6
5.1.5.1 Exception Records for General Exceptions 5–9
5.1.5.2 Exception Records for Unwind Exceptions 5–9
5.1.5.3 Exception Records for Signal Exceptions 5–9
5.1.6 Frame-Based Exception Handlers 5–10
5.1.7 Establishing Handlers 5–10
5.1.8 Raising Exceptions 5–11
5.1.8.1 Raising General Exceptions 5–12
5.1.8.2 Raising General Exceptions Using gentrap 5–12
5.1.8.3 Raising Unwind Exceptions 5–14
5.1.8.4 Raising Signal Exceptions 5–14
5.1.9 Search for and Invocation of Exception Handlers 5–14
5.1.9.1 Invocation Order for Exception Handlers 5–14
5.1.9.2 Handler Invocation and Arguments 5–17
5.1.10 Modification of Exception Records and Context by Handlers 5–18
5.1.11 Handler Completion and Return Value 5–19
5.1.11.1 Completion by Reraising the Exception 5–20
5.1.11.2 Completion by Continuing Thread Execution 5–20
5.1.11.3 Completion During Unwinding 5–20
5.1.11.4 Completion from Signal Exceptions 5–20
5.1.12 Other Considerations in Handling Exceptions 5–20
5.1.12.1 Exception Synchronization 5–21
5.1.12.2 Continuation from Exceptions 5–22
5.1.13 Exception and Signal Handling Coexistence 5–22
5.2 Unwinding 5–24
5.2.1 Overview of Unwinding 5–24
5.2.2 Types of Unwind Operations 5–25
5.2.2.1 General Unwind 5–25
5.2.2.2 Exit Unwind 5–25
5.2.3 Types of Unwind Invocations 5–26
5.2.3.1 Unwind Operations with No Active Exception 5–26
5.2.3.2 Unwind Operations During an Active Exception 5–26
5.2.4 Unwind Initiation 5–27
5.2.5 Unwinding from an Inserted Code Range 5–28
5.2.6 Multiply Active Unwind Operations 5–28
5.2.6.1 Nested Unwind 5–29
5.2.6.2 Colliding Unwind 5–29
5.2.7 Unwind Completion 5–29
5.2.8 Unwinding Coexistence with setjmp and longjmp 5–30

6 Stack Limits in Multithreaded Execution Environments

6.1 Stack Limit Checking 6–1
6.1.1 Stack Region Definitions 6–2
6.1.2 Methods for Stack Limit Checking 6–2
6.1.2.1 Implicit Stack Limit Checking 6–3
6.1.2.2 Explicit Stack Limit Checking 6–4
6.1.3 Stack Reserve Region Checking 6–5
6.2 Stack Overflow Handling 6–5

7 Procedure Invocations and Call Chains

7.1 Referencing a Procedure Invocation 7–1
7.2 Providing a Procedure Invocation Context 7–2
7.3 Walking the Call Chain 7–2

8 Procedure Descriptors

8.1 Procedure Descriptor Representation 8–1
8.1.1 Code Range Descriptors 8–3
8.1.2 Run-Time Procedure Descriptors 8–5
8.1.3 Examples 8–12
8.1.3.1 Multiple Entry Point Example 8–12
8.1.3.2 Instrumented Code Example 8–14
8.2 Procedure Descriptor Access Routines 8–16
8.3 Run-Time Generated Code 8–18

Index

Figures

3–1 Fixed Size Stack Frame Format 3–5
3–2 Variable Size Stack Frame Format 3–6
3–3 Register Save Area Layout 3–9
3–4 Register Save Area Example 3–10
4–1 In-Memory Homed Argument List Structure 4–4
5–1 Status Value Representation 5–5
5–2 Exception Record Format 5–7
5–3 Procedure Invocation Chain 5–16
8–1 Code Range Descriptor 8–4
8–2 Long Form Run-Time Procedure Descriptor 8–7
8–3 Short Form Run-Time Procedure Descriptor 8–8

Tables

2–1 General-Purpose Integer Registers 2–2
2–2 Floating-Point Registers 2–3
3–1 Procedure Properties Summary 3–2
4–1 Argument Item Locations 4–2
4–2 Unused Bits in Passed Data 4–6
4–3 Data Alignment Addresses 4–12
5–1 gentrap EXPT_CODE Values 5–13

About This Manual

This manual defines the requirements, mechanisms, and conventions that are used in the UNIX interface that supports procedure calls for HP Tru64 UNIX for Alpha systems. The standard defines the 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 Tru64 UNIX systems operating on Alpha hardware.

Audience

Although this manual primarily defines requirements for compiler and debugger writers, the information applies to procedure calling for all programmers at all levels of programming.

New and Changed Features

Changes or additions to information on the following topics have been made to this manual for the Version 5.1 release of Tru64 UNIX:

- Chapter 4:
  - Passing extended-precision complex values by reference — *Section 4.1.2*
  - Passing single- and double-precision complex values as two real values in two argument items — *Section 4.1.6.1*
  - Quadword alignment of global variables — *Section 4.2.1*
  - Granularity requirements of byte and word instructions — *Section 4.2.2*
- Chapter 5:

  Unwinding from an inserted code range — *Section 5.2.5*
- Chapter 8:
  - New fields in code range descriptors and procedure descriptors to define the context of any given instruction in a procedure — *Section 8.1.1* and *Section 8.1.2*
  - Examples showing use of code range descriptors and procedure descriptors — *Section 8.1.3*

Organization

This document includes eight chapters:

Chapter 1	Introduces the standard and provides definitions of terms used in the standard.
Chapter 2	Describes the fundamental concepts of the Tru64 UNIX calling standard for Alpha systems.
Chapter 3	Describes the aspects of the standard that deal with flow control.
Chapter 4	Discusses the passing and storage of data.
Chapter 5	Discusses how the standard relates to events outside the normal program flow.
Chapter 6	Discusses stack limit checking in multithreaded execution environments.
Chapter 7	Describes the mechanisms for functions that are needed to support procedure call tracing.
Chapter 8	Discusses procedure descriptors.

Related Documents

This Tru64 UNIX calling standard is a component of the larger Alpha Software Architecture and depends on standards and conventions not described in this document. These standards include:

- Object language (including link-time optimizations) and object file format
- Status values and message definition, formatting, and reporting
- Heap memory management and dynamic string management
- Multithread architecture
- Names and naming conventions

The following documents contain information related to this standard and the standards mentioned in the previous list:

- *Alpha Architecture Reference Manual*, 2nd Edition (Butterworth-Hinemann Press, ISBN:1-55558-145-5)
- *Programmer's Guide*
- *Assembly Language Programmer's Guide*
- *Guide to DECthreads*
- *Object File / Symbol Table Format Specification* (This manual is available in HTML and PDF formats on the Tru64 UNIX Documentation CD-ROM, V5.0 and higher; it is not available in hardcopy or in PostScript format.)

- *Compaq Portable Mathematics Library*
- *POSIX Conformance Document*
- *Information Technology – Portable Operating System Interface (POSIX) – Part 1: System Application Program Interface (API) [C Language] ISO/IEC 9945-1: 1990*
- *American National Standard for Information Systems Programming Language C – ANSI X3.159-1989* and its international equivalent, *ISO/IEC 9899*

Icons on Tru64 UNIX Printed Manuals

The printed version of the Tru64 UNIX documentation uses letter icons on the spines of the manuals to help specific audiences quickly find the manuals that meet their needs. (You can order the printed documentation from HP.) The following list describes this convention:

G	Manuals for general users
S	Manuals for system and network administrators
P	Manuals for programmers
R	Manuals for reference page users

Some manuals in the documentation help meet the needs of several audiences. For example, the information in some system manuals is also used by programmers. Keep this in mind when searching for information on specific topics.

The *Documentation Overview* provides information on all of the manuals in the Tru64 UNIX documentation set.

Reader's Comments

HP welcomes any comments and suggestions you have on this and other Tru64 UNIX manuals.

You can send your comments in the following ways:

- Fax: 603-884-0120 Attn: UBPG Publications, ZKO3-3/Y32
- Internet electronic mail: `readers_comment@zk3.dec.com`

  A Reader's Comment form is located on your system in the following location:

  `/usr/doc/readers_comment.txt`

Please include the following information along with your comments:

- The full title of the manual and the order number. (The order number appears on the title page of printed and PDF versions of a manual.)
- The section numbers and page numbers of the information on which you are commenting.
- The version of Tru64 UNIX that you are using.
- If known, the type of processor that is running the Tru64 UNIX software.

The Tru64 UNIX Publications group cannot respond to system problems or technical support inquiries. Please address technical questions to your local system vendor or to the appropriate HP technical support office. Information provided with the software media explains how to send problem reports to HP.

Conventions

This document uses the following typographical and symbol conventions:

`%` `$`	A percent sign represents the C shell system prompt. A dollar sign represents the system prompt for the Bourne, Korn, and POSIX shells.
`%` **`cat`**	Boldface type in interactive examples indicates typed user input.
`file`	Italic (slanted) type indicates variable values, placeholders, and function argument names.
[|] { | }	In syntax definitions, brackets indicate items that are optional and braces indicate items that are required. Vertical bars separating items inside brackets or braces indicate that you choose one item from among those listed.
`cat`(1)	A cross-reference to a reference page includes the appropriate section number in parentheses. For example, `cat`(1) indicates that you can find information on the `cat` command in Section 1 of the reference pages.

The following presentation conventions apply to this standard:

- Constants, data structures, and functional interfaces

Constants are represented symbolically with their values given at the point of definition in the standard. Data structures are defined in terms of the physical memory format of each structure. Functional interface syntax is presented in abstract form. Concrete language bindings for each constant, data structure, and functional interface are provided in system definition files external to the standard.

- Algorithms

  Algorithms are presented as a series of steps in standard American English.

- Numbering

  All numbers are represented as decimal values unless otherwise indicated. Nondecimal numbers are represented with the base name in parentheses following the number, for example, `1B(hex)`.

- Memory and register layouts

  Figures that represent memory or register layouts follow the convention that increasing addresses run from top to bottom and right to left. The most significant bits are on the left; the least significant bits are on the right.

- Code examples

  All code examples are supplied to clarify the concept under discussion. These examples do not necessarily reflect the optimized or properly scheduled code sequences that a compiler would generate. The assembly language syntax follows the conventions used in the *Assembly Language Programmer's Guide*.

- Record fields

  Record fields are referred to by using the name of the record or subrecord followed by a dot (.) and then the field name:

  record-name.subrecord-name.field

1 Introduction

This manual defines the requirements, mechanisms, and conventions used in the Tru64 UNIX interface that supports procedure calls for Tru64 UNIX for Alpha systems. The standard defines the 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 Tru64 UNIX systems operating on Alpha hardware.

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 to all points throughout the execution of standard-conforming user-mode code and must, therefore, be held constant at all times. 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; for example, 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

The specifications in this standard are presented in an "as if" manner; that is, all conformant code must behave as if the specifications have been met. This standard is designed so that additional link-time information can be utilized to optimize or even remove instructions in critical code paths and, as such, achieve higher performance levels.

In many cases, significant performance gains can be realized by selective use of nonstandard calls when the safety of such calls is known. Compiler writers are encouraged to make full use of such optimizations, but should make sure that procedures outside the compilation unit can proceed as if the standard were met.

The conventions specified in this standard are intended to exploit fully the architectural and performance advantages of the Alpha hardware. Some of these conventions are visible to the high-level language programmer and, therefore, might require source changes in high-level language programs when they are ported from other environments. Users should not depend on the properties of the Alpha architecture to achieve source-level compatibility and portability between Tru64 UNIX for Alpha systems and other UNIX environments, except indirectly through high-level language facilities that are portable across architectures.

1.1 Applicability

This manual defines the rules and conventions that govern the **native user-mode run-time environment** on Tru64 UNIX systems running on Alpha hardware. The standard is applicable to all products in native user mode on the Tru64 UNIX operating system.

This standard applies to the following:

- All externally callable interfaces written in standard system software
- All intermodule calls to major software components
- All external procedure calls generated by language processors without benefit of interprocedural analysis or permanent private conventions, such as those for language support run-time library routines

1.2 Architectural Level

This Tru64 UNIX calling standard defines an **implementation-level run-time software architecture** for Tru64 UNIX operating systems running on Alpha hardware.

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

Compilers and run-time libraries may provide additional support for these capabilities through interfaces that are more appropriate for compilers and applications. This standard neither prohibits nor requires such additional interfaces.

1.3 Goals

In general, this Tru64 UNIX calling standard promotes the highest degree of performance, portability, efficiency, and consistency in the interface between called procedures in the Tru64 UNIX environment.

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

The goals of the Tru64 UNIX calling standard are to:

- Include capabilities specifically for lower-level components (such as assembler routines) that cannot be invoked from the high-level languages.
- Allow the calling program and called procedure to be written in different languages. The standard is designed to reduce the need to use language extensions for mixed-language programs.
- Contribute to the writing of error-free, modular, and maintainable software and promote effective sharing and reuse of software modules.
- Provide the programmer with control over the fixing and reporting of exception conditions and with management of the flow of control when various types of exception conditions occur.
- Add no space or time overhead to procedure calls and returns that do not establish exception handlers. The standard is designed to minimize the time overhead for establishing handlers at the cost of increasing time overhead when exceptions occur.
- Be optimized for newer, more complex compilation techniques, such as interprocedural analysis and link-time code transformations. However, the standard is designed to require no such mechanisms for correctness.
- Provide support for a multilanguage, multithreaded execution environment.
- Provide an efficient mechanism for calling lightweight procedures that do not need or want to pay the overhead of setting up a stack call frame. (Procedures are referred to as lightweight because of their expedient way of saving the call context.)
- Provide 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. (Procedures that incur costs by storing the call context in memory are referred to as heavyweight.)

This calling standard is designed to allow 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 should not require a recompilation of its callers.

- Provide condition handling, traceback, and debugging for lightweight procedures that do not have a stack frame.
- Make efficient and effective use of the Alpha hardware architecture.
- Minimize the cost of procedure calls.
- Support a 64-bit address user-mode environment.

1.4 Requirements

The Tru64 UNIX calling standard was developed with the following requirements:

- All Alpha platforms must be able to implement the standard.
- Non-Digital compiler writers must be able to implement the standard.
- The standard must not require any complex compilation techniques (such as link-time code movement) for correctness.

1.5 Definitions

The following terms are used in the Tru64 UNIX calling standard:

address	A 64-bit value used to denote a position in memory.
argument list	A vector of quadword entries that represents a procedure parameter list and possibly a function value.
bound procedure	A type of procedure that requires knowledge at run time of a dynamically determined larger enclosing scope to execute correctly.
call frame	The body of information that a procedure must save to allow it to return properly to its caller. A call frame can exist on the stack or in registers. Optionally, a call frame can contain additional information required by the called procedure.
condition	See **exception condition**.

descriptor	A mechanism for passing parameters, where the address of the descriptor is an entry in the argument list. The descriptor contains the parameter's address, data type, and size as well as additional information needed to describe fully the data passed.
exception condition	An exceptional condition in the current hardware and/or software state that should be noted or fixed. The existence of this condition causes an interrupt in program flow and forces execution of out-of-line code. Such an event may be caused by exceptional hardware states (for example, arithmetic overflows or memory access control violations) or by actions performed by software (for example, subscript range checking, assertion checking, or asynchronous notification of one thread by another). While the normal control flow is interrupted by an exception, the program flow is said to be in the **active** state.
exception handler	A procedure designed to handle exception conditions when they occur during the execution of a thread.
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 directly reflects an exception 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.
image	A collection of compiled modules that are combined by a linker into a form that can be loaded for execution.
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 higher-level language constructs. Such procedures are not intended to be explicitly called from a user program.
library procedure	A procedure explicitly called using the equivalent of a call statement or function reference. Such procedures are usually language-dependent.
natural alignment	An attribute of certain data types that refers to the placement of the data so that the lowest addressed byte 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 descriptor	A set of information about the properties of a procedure. This information is contained in data structures at run time to enable exception handling and unwinding to function properly.
procedure value	An address value that represents a procedure value. This value is the address of the first instruction of the procedure to be executed.
process	An address space containing at least one thread of execution. Selected security and quota checks are performed on a per-process basis.

This standard applies to the execution of multiple threads within a process. (An operating system that only provides a single thread of execution per process is considered a special case of a multithreaded system; that is, one where the maximum number of threads per process is 1.)

reference	A mechanism for passing parameters, where the calling program provides the parameter's address in the argument list.
sharable image	An image that can be shared by multiple processes. On Tru64 UNIX, a single copy of the image can be included simultaneously at different addresses in multiple using processes. Such an image is said to be position-independent.
signal	A POSIX-defined concept used to cause out-of-line execution of code.
standard call	A transfer of control to a procedure by any means that presents the called procedure with the environment defined by this standard and does not place additional restrictions, not defined in this standard, on the called procedure.
standard conforming procedure	A procedure that adheres to all the relevant rules set forth in the Tru64 UNIX calling standard.
thread, or thread of execution	An entity scheduled for execution on a processor. In language terms, a thread is a computational entity utilized by a program unit such as a task, procedure, or loop. All threads executing within the same process share the same address space and other process context, but have unique per-thread hardware contexts that include machine registers such as program counters, process status, and stack pointers. 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 these user-mode process threads.

thread-safe code	A property of code compiled in such a way as to ensure that 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 way.
undefined	Operations or behavior for which there is no directing algorithm used across all implementations that support the Tru64 UNIX calling standard. Such operations may or may not be well defined for a single implementation, but remain undefined with reference to this standard. The actions of undefined operations may not be required by standard-conforming procedures.
unpredictable	Any results of an operation that cannot be guaranteed across all operating system implementations of the Alpha architecture calling standard. Regardless of whether these results are well-defined for a specific implementation of the Alpha calling standard, they remain unpredictable with reference to all implementations of the standard.
	Therefore, all results caused by operations defined in the Tru64 UNIX implementation of the calling standard, but not specified as part of the calling standard are considered unpredictable. Standard-conforming procedures cannot depend on unpredictable results.

2 Tru64 UNIX Concepts

This chapter describes some fundamental concepts of the Tru64 UNIX calling standard. The topics discussed are as follows:

- Address representation (Section 2.1)
- Procedure representation (Section 2.2)
- Register usage conventions (Section 2.3)
- Register names (Section 2.4)
- Program image layout (Section 2.5)

2.1 Address Representation

Values that represent addresses are 64 bits in size.

2.2 Procedure Representation

One of the distinguishing characteristics 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 Tru64 UNIX calling standard, a simple (not bound) procedure value is defined as the address of the first instruction of that procedure's entry code. (See Section 3.2.6.)

A bound procedure value is defined as the address of the first instruction of an instruction sequence that establishes the correct execution context for the bound procedure.

Procedures in the Tru64 UNIX calling standard are associated with a set of information called a procedure descriptor. This information describes various aspects of the procedure's code that are required for correct and robust exception handling. The exception processing described by this standard is based on the assumption that any given program counter value can be mapped to an associated procedure descriptor that describes the currently active procedure.

2.3 Register Usage Conventions

This section describes the usage of the Alpha hardware integer and floating-point registers.

2.3.1 Integer Registers

Table 2–1 describes the Alpha hardware integer registers.

Table 2–1: General-Purpose Integer Registers

Register	Description
$0	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 can be modified by the called procedure without being saved and restored.
$1 – $8	Conventional scratch registers. In a standard call, these registers can be modified by the called procedure without being saved and restored.
$9 – $14	Conventional saved registers. If a standard-conforming procedure modifies one of these registers, it must save and restore it.
$15	Stack frame base (FP) register. For procedures with a run-time variable amount of stack, this register is used to point at the base of the stack frame (fixed part of the stack). For all other procedures, this register has no special significance. If a standard-conforming procedure modifies this register, it must save and restore it.
$16 – $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 can be modified by the called procedure without being saved and restored.
$22 – $25	Conventional scratch registers. In a standard call, these registers can be modified by the called procedure without being saved and restored.
$26	Return address (RA) register. In a standard call, the return address must be passed and returned in this register.
$27	Procedure value (PV) register. In a standard call, the procedure value of the procedure being called is passed in this register. (See Section 2.2.) In a standard call, this register can be modified by the called procedure without being saved and restored.
$28	Volatile scratch register. The contents of this register are always *unpredictable* after any external transfer of control to or from a procedure. This unpredictable nature applies to both standard and nonstandard calls. This register can be used by the operating system for external call fixing, autoloading, and exit sequences.

Table 2–1: General-Purpose Integer Registers (cont.)

Register	Description
$29	Global pointer (GP) register. For a standard-conforming procedure, this register must contain the calling procedure's global offset table (GOT) segment pointer value at the time of a call and must contain the calling procedure's GOT segment pointer value or the called procedure's GOT segment pointer value upon return. This register must be treated as scratch by the calling procedure. (See Section 2.5 and Section 3.2.2 for details.)
$30	Stack pointer (SP) register. 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. See Section 3.2.1 for details about its usage and alignment.
$31	ReadAsZero/Sink register. This register is defined in the hardware to be binary zero as a source operand or sink (no effect) as a result operand.

2.3.2 Floating-Point Registers

Table 2–2 describes the Alpha hardware floating-point registers.

Table 2–2: Floating-Point Registers

Register	Description
$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 can 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 can be modified by the called procedure without being saved and restored.
$f2 – $f9	Conventional saved registers. If a standard-conforming procedure modifies one of these registers, it must save and restore it.
$f10 – $f15	Conventional scratch registers. In a standard call, these registers can be modified by the called procedure without being saved and restored.
$f16 – $f21	Argument registers. In a standard call, up to six floating-point arguments can be passed by value in these registers. In a standard call, these registers can be modified by the called procedure without being saved and restored.

Table 2–2: Floating-Point Registers (cont.)

Register	Description
$f22 – $f30	Conventional scratch registers. In a standard call, these registers can be modified by the called procedure without being saved and restored.
$f31	ReadAsZero/Sink register. This register is defined in the hardware to be binary zero as a source operand or sink (no effect) as a result operand.

2.4 Register Names

A few special register names appear in uppercase letters. This register naming convention is not necessarily followed by compiler or assembler tools that are used on Tru64 UNIX systems. However, a simple name substitution can convert from the notation used here to the appropriate convention. For example, the following code fragment might be used in conjunction with the standard C preprocessor as a prelude to the examples in this text:

```
#define FP    $15
#define RA    $26
#define PV    $27
#define GP    $29
#define SP    $30
```

2.5 Program Image Layout

The Tru64 UNIX calling standard defines only some aspects of an executable image. One basic concept that is defined involves program image layout, which permits optimal access to static data.

A hardware architecture in which instructions cannot contain full virtual addresses can be referred to as a base register architecture. The Alpha architecture is such an 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 some register which contains that address (usually referred to as a **base register**). Base registers for external program segments, either data or code, are usually loaded indirectly through a program segment of address constants.

To optimize this base register access method, this standard requires each image that makes up an executable program to have zero or one **global offset table** (GOT). This global offset table can be further divided into multiple GOT segments. Together, the linker and the compilers arrange for various static data to be collected together into a minimal number of these GOT segments (typically one per image). During program execution, the GP (global pointer) register will contain a pointer into the appropriate GOT segment so that all references therein can utilize a single base register. (For more information, see Section 3.2.2.)

During the compilation process, a compiler generates object language to designate data as belonging in a GOT segment. No single procedure is allowed to provide more than 64KB of data to a GOT segment. The linker pools these contributions to form the GOT segments. Typically, routines in several compilation units can share the same pointer into the GOT. In fact, if only one GOT segment is needed and will not exceed the 64KB addressing maximum, all routines within an image can use the same GOT pointer. Consequently, the GP register can be loaded once and then used by many routines to improve performance.

3 Flow Control

This chapter contains descriptions of those aspects of the calling standard that deal with the flow of control in a program. This chapter does not discuss data manipulation. That topic is discussed in Chapter 4.

The following sections describe:

- Procedure types (Section 3.1)
- Transfer of control (Section 3.2)

3.1 Procedure Types

This Tru64 UNIX calling standard defines three basic procedures types. A compiler may choose which type to generate based on the requirements of the procedure in question. The standard procedure types are:

- Stack frame procedure

  A procedure that maintains its caller's context on the stack
- Register frame procedure

  A procedure that maintains its caller's context in registers
- Null frame procedure

  A procedure that executes in the context of its caller

Some procedures maintain their call frame on the stack; others maintain their call frame entirely in registers, although they may use the stack. Simple procedures do not necessarily maintain any call frame, but instead execute completely in the context of their caller. The calling procedure does not need to distinguish among these cases.

3.1.1 Procedure Descriptor Overview

Every procedure, other than a null procedure described in Section 3.1.4, must have a set of information associated with it that describes which type of procedure it is and what characteristics it has. This set of information, called a **procedure descriptor**, can be thought of as a single structure, although physically it is implemented by several structures. (See Chapter 8.) The procedure descriptor structure is used to interpret the call chain at any point during a thread's execution. The structure is normally built at compile time and is not generally accessed at run time except in support of exception processing or other rarely used code execution.

Table 3–1 briefly summarizes the properties of a procedure that can be determined from its associated procedure descriptor. (Some fields apply to only certain kinds of procedures.) For a complete description of procedure descriptors, see Section 8.1.

Table 3–1: Procedure Properties Summary

Procedure Property	Description
`PDSC_FLAGS_REGISTER_FRAME`	Indicates a register (or null) frame procedure rather than a stack frame procedure.
`PDSC_FLAGS_BASE_REG_IS_FP`	Indicates that register $15 is used as a frame pointer rather than as a preserved register.
`PDSC_FLAGS_HANDLER_VALID`	Indicates that there is an associated exception handler.
`PDSC_FLAGS_EXCEPTION_MODE`	Indicates the error-reporting behavior expected from certain called mathematical library routines.
`PDSC_FLAGS_EXCEPTION_FRAME`	Indicates that an operating system exception frame is included in the procedure's frame.
`PDSC_FLAGS_ARITHMETIC_SPECULATION`	Indicates that arithmetic traps (`SIGFPE`) occurring in this procedure should not be delivered. This procedure was compiled with speculative execution optimization applied to arithmetic operations. Some arithmetic instructions have been moved backward, preceding conditional branches that used to control their effects. Traps caused by these moved instructions should be ignored.

Table 3–1: Procedure Properties Summary (cont.)

Procedure Property	Description
PDSC_CRD_MEMORY_SPECULATION	Indicates that memory traps (SIGSEGV, SIGBUS) occurring in this procedure should not be delivered. This procedure was compiled with speculative execution optimization applied to memory reference operations. Some memory reference instructions have been moved backward, preceding conditional branches that used to control their effects. Traps caused by these moved instructions should be ignored.
PDSC_RPD_RSA_OFFSET	Specifies an offset from the stack pointer (SP) or frame pointer (FP) register to the register save area.
PDSC_RPD_IMASK	Indicates a bit mask for the general registers that are saved in the stack.
PDSC_RPD_FMASK	Indicates a bit mask for the floating-point registers that are saved in the stack.
PDSC_RPD_ENTRY_RA	Indicates the register that contains the return address at the time of a call.
PDSC_RPD_SAVE_RA	Indicates the register in which the return address is saved when it is not saved on the stack.
PDSC_RPD_FRAME_SIZE	Specifies the size of the fixed part of the stack frame.
PDSC_RPD_SP_SET	Specifies the offset from the beginning of the procedure to the instruction that changes the stack pointer.
PDSC_RPD_ENTRY_LENGTH	Specifies the length of the procedure prologue.
PDSC_CRD_BEGIN_ADDRESS	Specifies the address of the first instruction and entry point of the procedure.
PDSC_RPD_HANDLER_ADDRESS	Specifies the address of an associated exception handling procedure.
PDSC_RPD_HANDLER_DATA	Specifies supplementary per-procedure data to be passed to an associated exception handler.

3.1.2 Stack Frame Procedure

A **stack frame procedure** is one that allocates space for and saves its caller's context on the stack. This type of procedure is sometimes called a "heavyweight procedure" because of the cost of storing this context in memory. These procedures can save and restore registers and may make standard calls to other procedures.

The stack frame of this type of 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 be careful to recognize situations that can effectively cause a variable part of the stack to exist in nonintuitive ways; for example, when a called routine uses the stack as a means to return certain types of function values. (See Section 4.1.7 for details.)

If such a situation exists, a compiler must choose to use a variable-size stack frame procedure when compiling the caller so that any unwind operations can be performed correctly.

3.1.2.1 Stack Frame Format

Even though the exact contents of a stack frame are determined by the compiler, there are certain properties common to all stack frames. There are two basic types of stack frames: fixed-size and variable-size. Some parts of the stack frame are optional and occur only as required by the particular procedure. In the figures, brackets ([ ]) surrounding a field's name indicate that the field is optional.

Figure 3–1 shows the format of the stack frame for a procedure with a fixed amount of stack. This format uses SP as the stack base register (that is, `PDCS_FLAGS_BASE_REG_IS_FP` is 0). In this case, $15 is simply another saved register and has no special significance.

Figure 3–1: Fixed Size Stack Frame Format

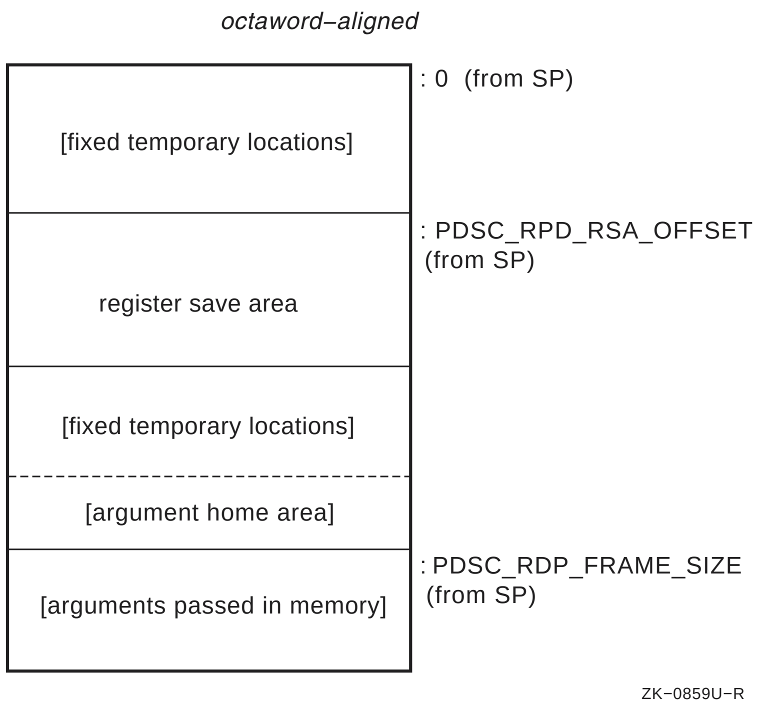
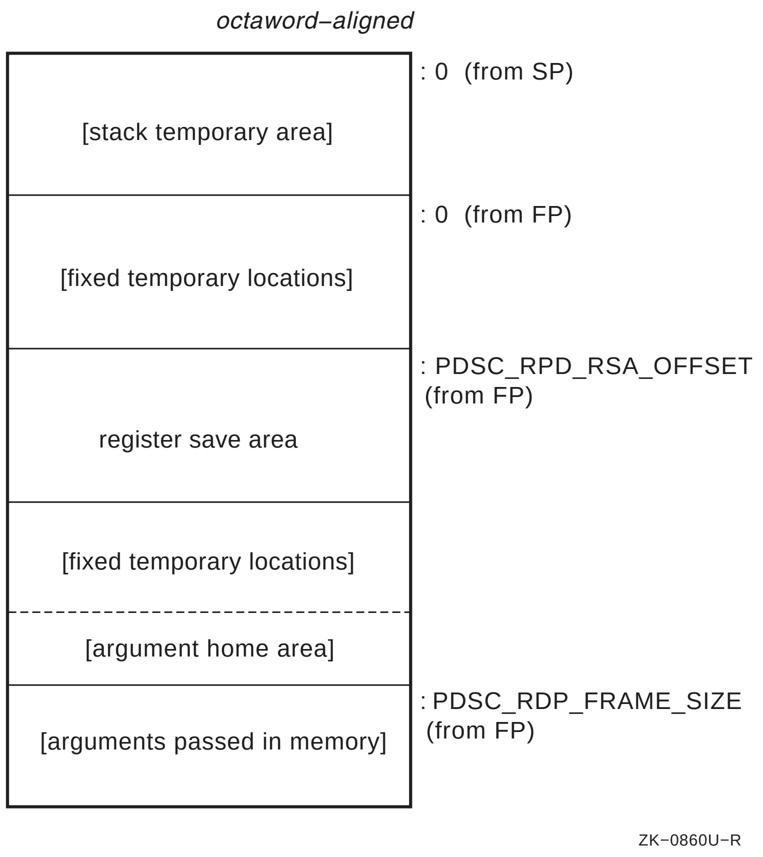

octaword–aligned
: 0 (from SP)
[fixed temporary locations]
: PDSC_RPD_RSA_OFFSET (from SP)
register save area
[fixed temporary locations]
[argument home area]
: PDSC_RDP_FRAME_SIZE (from SP)
[arguments passed in memory]
ZK−0859U−R

Figure 3–2 shows the format of the stack frame for a procedure with a varying amount of stack. The format uses FP as the stack base register; that is, `PDSC_FLAGS_BASE_REG_IS_FP` is 1.

Figure 3–2: Variable Size Stack Frame Format



octaword–aligned
[stack temporary area]
: 0 (from SP)
[fixed temporary locations]
: 0 (from FP)
register save area
: PDSC_RPD_RSA_OFFSET (from FP)
[fixed temporary locations]
[argument home area]
[arguments passed in memory]
: PDSC_RDP_FRAME_SIZE (from FP)
ZK−0860U−R

In both cases, the portion of the stack frame designated by PDSC_RPD_FRAME_SIZE must be allocated and initialized by the entry code sequence of a called procedure with a stack frame.

Fixed temporary locations are optional sections of the stack frame that contain language-specific locations required by the procedure context of some high-level languages. These locations might include, for example, register spill areas, language-specific exception handling contexts (such as language dynamic exception handling information), or fixed temporary locations.

If a compiler chooses, the fixed temporary locations adjacent to the area pointed to by the frame base register added to the value of PDSC_RPD_FRAME_SIZE can be used for a special purpose referred to as the argument home area.

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 that all arguments can be addressed as a contiguous array. This area can also be used to store arguments that are passed in registers if an address for such an argument must be generated. Generally, 6 or 12 contiguous quadwords of stack storage are allocated by the called procedure for this kind of storage. (See Section 4.1.3 for details.)

The **register save area** is a set of consecutive quadwords where the current procedure saves and restores registers. The register save area begins at the location pointed to by the frame base register (as indicated by `PDSC_FLAGS_BASE_REG_IS_FP`) added to the value of the contents of `PDSC_RPD_RSA_OFFSET`. The result must be a quadword-aligned address. The set of registers saved in this area contains the return address followed by the registers specified in the procedure descriptor by `PDSC_FLAGS_IMASK` and `PDSC_FLAGS_FMASK`. The details of how to lay out and populate the register save area are described in Section 3.1.2.2.

A compiler can use the **stack temporary area** for storage of fixed local variables, such as constant-sized data items, program state information, and dynamically sized local variables. The stack temporary area can also be used for dynamically sized items with a limited lifetime, such as a dynamically sized function result or string concatenation that cannot be directly stored in a target variable. When a procedure uses this area, the compiler must keep track of its base and reset SP to that base in order to reclaim storage used by temporaries.

The high-address end of the stack frame is defined by the value stored in `PDSC_RPD_FRAME_SIZE` added to the contents of the SP or FP register, as indicated by `PDSC_FLAGS_BASE_REG_IS_FP`. The high-address end is used to determine the value of the SP register for the predecessor procedure in the call chain.

3.1.2.2 Register Save Area

The layout of the frame of a stack frame procedure contains a substructure called the **register save area**. This section describes how this area is defined and populated.

All registers saved in the variable portion of the register save area must have the corresponding bit set to 1 in the appropriate procedure descriptor register save mask. This bit must be set to 1 even if the register is not a member of the set of registers required to be saved across a standard call. If the bit is not set to 1, the offsets within the save area cannot be calculated correctly.

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

- The return address is saved at the lowest address of the register save area, offset 0.

- All saved integer registers ,as indicated by the corresponding bit in `PDSC_RPD_IMASK` being set to 1, are stored, in register-number order, in consecutive quadwords beginning at offset 8 of the register save area.
- All saved floating-point registers , as indicated by the corresponding bit in `PDSC_RPD_FMASK` being set to 1, are stored, in register-number order, in consecutive quadwords following the saved integer registers.

Note

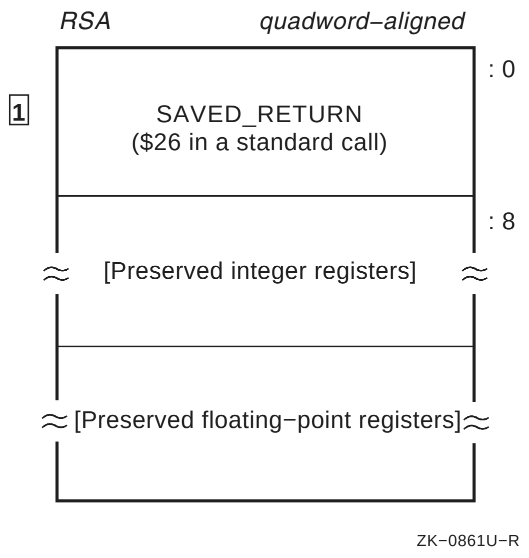
A floating-point register saved in the stack is stored as a 64-bit exact image of the register; that is, no bit reordering is done in the process of moving the data to or from memory. Compilers must use an `stt` instruction to store the register regardless of the floating-point type.

This behavior is required so that an unwind routine can properly restore the floating-point registers without more complete type information.

A standard-conforming procedure that utilizes a register save area must save the return address register at offset 0 in the register save area. There is no corresponding bit in the register save mask for this register slot.

Figure 3–3 shows the layout of the register save area.

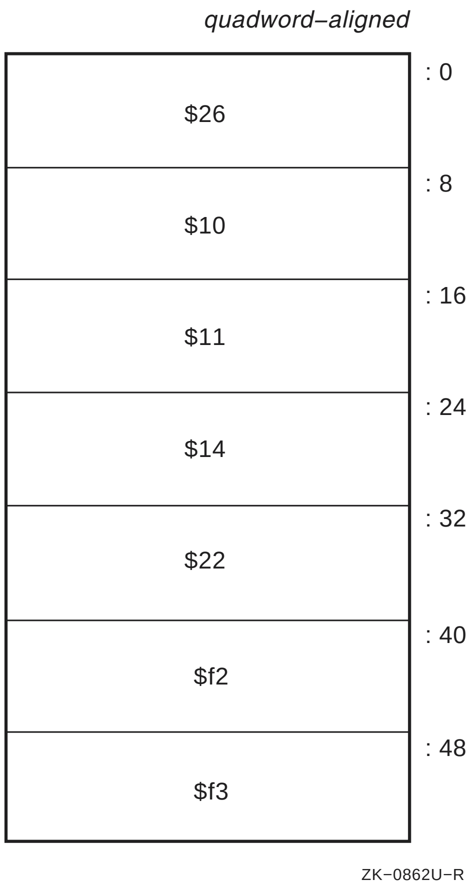
Figure 3–3: Register Save Area Layout



$\boxed{1}$ `RSA.SAVED_RETURN` is the contents of the return address register.

For example, if registers $10, $11, $14, $22, $f2, and $f3 are saved by a standard-conforming procedure, the `PDSC_RPD_IMASK` value is 00404C00 (hex) and the `PDSC_RPD_FMASK` is 0000000C (hex). The register save area for such a procedure is packed as shown in Figure 3–4.

Figure 3–4: Register Save Area Example



3.1.3 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" because of the relatively fast way it saves the call context.

Such a procedure cannot save and restore nonscratch registers. Because a procedure without a stack frame must, therefore, use scratch registers to maintain the caller's context; such a procedure cannot make a standard-conforming 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 the use of appropriate agreements with callers of the lightweight procedure as well as 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.

Agreements such as these can be made by convention (as in the case of language-support routines in the run-time library) or by interprocedural analysis. Calls employing such agreements are, however, not standard calls, and might not be fully supported by a debugger because it might not be able to find the contents of the preserved registers, for example.

Because such agreements must be permanent for upward compatibility of object code, lightweight procedures should, in general, follow the normal restrictions.

3.1.4 Null Frame Procedure

A **null frame procedure** is a simple case of a register frame procedure. The null frame procedure has the following characteristics:

- The entry return address register is $26 (`PDSC_RPD_ENTRY_RA` = 26).
- The return address is not saved in any other register (`PDSC_RPD_SAVE_RA` = `PDSC_RDP_ENTRY_RA`).
- No stack space is allocated (`PDSC_RPD_SP_SET` = 0 and `PDSC_RPD_FRAME_SIZE` = 0).
- As a result of these characteristics, the prologue requires no instructions (`PDSC_RPD_ENTRY_LENGTH` = 0).
- There is no associated exception handler (`PDSC_RPD_HANDLER_ADDRESS` = 0).

This special case of a register frame procedure is of interest because it has an associated special-case procedure descriptor representation. (See Section 8.1 for information about procedure descriptor representation.)

3.2 Transfer of Control

A standard-conforming procedure call can use any sequence of instructions that presents the called routine with the required environment. (See the standard call definition in Section 1.5.) However, the majority of standard-conforming external calls is performed with a common sequence of instructions and conventions. This common set of call conventions is so pervasive that it is included as part of this standard in Section 3.2.1.

This calling standard has been designed so that the same instruction sequence can be used to call each different type of procedure; that is, the caller does not have to know which type of procedure is being called.

3.2.1 Call Conventions

The following call conventions describe the rules and methods used to communicate certain information between the caller and the called procedure during invocation and return:

procedure value The calling procedure must pass the procedure value of the called procedure to the called procedure. This value can be a statically or dynamically bound procedure value. To pass this value, the calling procedure must load $27 with the procedure value before control is transferred to the called procedure. (See Section 2.2 for a description of procedure values.)

When a target routine is not loaded in memory at the beginning of execution of a main program or shared image, the procedure value used by a caller of that routine generally addresses some kind of stub or jacket routine. The purpose of a stub or jacket routine is to perform the loading of the actual target routine. The call is completed after this load operation.

When control actually reaches the target routine entry point, $27 must contain the actual procedure value of the newly loaded routine as if no intermediate processing had occurred. Subject to these constraints, the PV register can be used freely by the stub/jacket code as a temporary register during its own execution.

return address The calling procedure must pass the address to which control must be returned to the called procedure 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. A standard-conforming procedure must treat this register as preserved. For a standard call, the return address is passed and returned in the return address register, $26.

argument list

The argument list is an ordered set of zero or more **argument items**, which together include a logically contiguous structure known as an **argument item sequence**. In practice, this logically contiguous sequence is mapped to registers and memory in a fashion that produces a physically discontiguous argument list. In a standard call, the first six items are passed in registers $16 – $21 and/or registers $f16 – $f21. (See Section 4.1.2 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).

function result

If a standard-conforming procedure is a function and the function result is returned in a register, the result is returned in $0, $f0, or $f0 – $f1. Otherwise, the function result is returned using the first argument item or else dynamically, as defined in Section 4.1.7.

stack usage

The stack pointer (SP) must at all times denote an address that has octaword alignment. (This restriction has the side effect that the in-memory portion of the argument list, if any, will start on an octaword boundary.) Note that the stack grows toward lower addresses. During a procedure invocation, SP can never be set to a value that is 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 (if any) that is passed in memory, belong to the calling procedure. Because they are part of the calling procedure, they should not be read or written by the called procedure, except as specified by indirect arguments or language-controlled up-level references.

The SP value might be used by the hardware when raising exceptions and asynchronous interrupts. It must be assumed that the contents of the stack below the current SP value and within the stack for the current thread are continually and unpredictably modified, as specified in the *Alpha Architecture Reference Manual*, and as a result of asynchronous software actions.

3.2.2 Linkage

When a standard procedure is called, the caller must provide the procedure value (code address) of the called procedure in $27 so the called procedure can compute the address of the global offset table (GOT) segment. (See Section 2.5 for information about the global offset table.)

If access to a GOT segment is required, two instructions in the called procedure's prologue will compute the GOT pointer value (GP register contents) using `ldah` and `lda` instructions together with the passed $27 value. Because the `ldah/lda` pair can generate addresses only within 2 GB of $27, the code and GOT must be within ± 2 GB of each other. Typically these instructions are the first two in the procedure. (In certain cases, at link time these two instructions may be replaced by `nop` instructions, skipped, or removed if the linker is able to determine that they are redundant.) The resultant pointer into the GOT segment is placed in the GP register. This pointer can then be used by the called procedure as a base register to address locations in the GOT.

Because a standard-conforming calling procedure must assume that the GP register value is destroyed across a call but must itself return it with the correct value, the code following the call must reestablish its value before further accesses to the GOT or by the time it returns from the procedure. (See Section 2.3.1 for information on integer registers.)

The following list describes some ways to reestablish the GP register value:

- The caller can use an `ldah/lda` sequence to compute the correct GP value from $26 because a standard call returns with $26 pointing at a known code address (the return address).
- The caller can save the GP value in a register that is preserved across the call and move it back into the GP register after the call.
- The caller can save the GP value in its stack frame and reload it after the call.

In summary, a standard-conforming call provides the procedure value (code address) to the called procedure (in case it needs to compute a new GP value) and provides its own GP value to the called procedure (in case it shares the GOT segment). Furthermore, upon return from the called procedure, the GP value must be restored (in case the called procedure did not share the same GOT segment).

3.2.3 Link-Time Optimization

The design of this calling standard assumes and expects that the normal call conventions described here will be improved by certain optimizations performed at link time in response to compiler-provided control information. However, the specified calling conventions will behave correctly even in the absence of link-time optimization.

For many calls, the called procedure shares the same GOT segment with the caller. In these cases, the GP register is already valid for the called procedure at the time of the call. Several optimizations are possible in these important cases. When one procedure calls another that shares the same GOT segment and the first two instructions of the called procedure establish the GP, the `ldah/lda` pair can be skipped. Because the procedure's code address has no other use, the caller does not need to provide it in $27. If the called procedure's GP value that is returned in the GP register is shared with the calling procedure, the caller does not need to reestablish the GP register contents.

The following code fragment shows a typical standard call:

```
ldq    $27,target_ptr(GP)   #Load procedure value (entry address)
jsr    $26,($27)            #Call with return address in $26
ldah   GP,fix_hi($26)       #Reload GP value
lda    GP,fix_lo(GP)
```

Note that other instructions may be scheduled among the ones shown here.

If the linker optimizes the call, it can be transformed to look like the following code fragment:

```
bsr    $26,target+8         #Make the call
```

The instructions that are no longer needed can be replaced by `nop` instructions, or deleted and compressed out. Depending on the optimizations performed and whether the `ldah/lda` pair at the target is moved, removed, or reduced to just `lda`, the call may or may not load $27, may execute a `jsr` or a `bsr`, and may go to `target` or `target+8`.

3.2.4 Calling Computed Addresses

Most calls are made to a fixed address whose value is already determined by the time the program starts execution. There are, however, certain cases that cause the exact address not to be determined until the code is actually executed. In these cases, the procedure value representing the procedure to be called will be computed in a register.

For example, suppose $4 contains such a computed procedure value (simple or bound). The following code fragment calls the procedure that $4 describes:

```
mov    $4,$27               #Move the procedure value to $27.
jsr    $26,($4)             #Call the entry address.
ldah   GP,fix_hi($26)       #Reload the GP register.
lda    GP,fix_lo(GP)
```

3.2.5 Bound Procedure Values

There are two distinct classes of procedures:

- Simple procedures
- Bound procedures

A **simple procedure** is one that does not need direct access to the stack of its execution environment. A **bound procedure** is one that must have direct access to the stack of its execution environment, typically so that it can reference an up-level variable or perform a nonlocal goto. Simple procedures and bound procedures have associated procedure descriptors, as described earlier in this chapter.

Bound procedure values are designed for multilanguage use. They allow callers of procedures to use common code to call both bound and simple procedures.

When a bound procedure is called, the caller must pass some kind of pointer to the called code to allow it to reference its up-level environment. Typically, such a pointer is the frame pointer for that environment, though many variations are possible. When the caller is itself executing within that outer environment, it usually can make such a call directly to the code for the nested procedure without recourse to any additional mechanism. 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 special bound procedure is created so the nested procedure can be called in the same way as a simple procedure.

The procedure value of a bound procedure is defined as the address of the first instruction of a sequence of instructions that establishes the proper environment for the bound procedure and then transfers control to that procedure.

One direct scheme for constructing a jacket to a bound procedure so it can be called like a simple procedure is to allocate a sequence of instructions and data on the stack and use the address of those instructions as the procedure value. For example, suppose that a bound procedure named `proc` expects its static link to be passed in $1. The following code fragment shows a suitable sequence of instructions for the call:

```
ldq   $1,24($27)         #Fetch the up-level pointer to $1
ldq   $27,16($27)        #Fetch the address of the bound
                         #   procedure code
```

```
jmp   ($27)              #Transfer to the bound procedure
nop                      #Include a filler to align following
                         #   data
```
quadword-holding-procedure-code-address
quadword-holding-static-link

Note that this sequence can only be created by code that is executing within the context of the containing procedure so that the appropriate frame pointer value is known and can easily be incorporated in the sequence illustrated. The lifetime of this sequence is, of course, limited to the lifetime of the stack frame in which it is allocated.

After creating the jacket instructions, it is necessary to execute an `imb` instruction before executing them to assure instruction cache coherence, as described in the *Alpha Architecture Reference Manual*. It might also be necessary to make the stack segment executable, for example, by using the `mprotect()` system call. (See the `mprotect`(2) reference page.)

3.2.6 Entry and Exit Code Sequences

The following sections describe the steps that must be executed in procedure entry and exit sequences. These conventions must be followed in order for the call chain to be well-defined at every point during thread execution.

Except as noted, the exact instruction sequences are not specified; any instruction sequence that produces the defined effects is legal.

3.2.6.1 Entry Code Sequence

Because the value of the PC defines the currently executing procedure, all properties of the environment specified by a procedure's descriptor must be valid before the first instruction after the procedure prologue (as defined by `PDSC_RPD_ENTRY_LENGTH`) is executed. In addition, none of the properties specified in the calling procedure's descriptor may be invalidated before the called procedure becomes current. Thus, until the procedure becomes current, all entry code must adhere to the following rules:

- All registers specified by this standard as saved across a standard-conforming call must contain their original (at entry) contents.
- The register designated by `PDSC_RPD_ENTRY_RA` ($26 in a standard call) must contain its original (at entry) contents. This requirement also applies to nonstandard procedures to allow for proper unwinding.
- No standard calls can be made.

Note

If an exception is raised or an exception occurs in the prologue of a procedure, that procedure's exception handler, if any, will not be invoked because the procedure is not yet current. Thus, when a procedure has an exception handler, compilers must not move into the procedure prologue any code that might cause an exception which would be handled by that same handler.

When a procedure is called, the code at the entry address must do the following:

- Synchronize, as needed, any pending exceptions caused by instructions issued by the caller.
- Save the caller's context.
- Make the called procedure current by executing the last instruction of the procedure prologue.

These actions involve the following steps, performed in the specified order:

1. Compute and load the procedure's GP value using the passed-in-procedure (code address) value in $27. (This code can appear elsewhere in the prologue, but provides more opportunities for linker optimizations if it appears first.)
2. If stack space is allocated (`PDSC_RPD_FRAME_SIZE` is not 0), set register SP to SP – `PDSC_RPD_FRAME_SIZE`.

   After any necessary calculations and stack limit checks, this step must be completed in exactly one instruction that modifies SP. This instruction must be the one specified by `PDSC_RPD_SP_SET`.
3. For a stack frame procedure (`PDSC_FLAGS_REGISTER_FRAME` is 0), do both of these steps. (There is no requirement as to which step occurs first.)
   – Store the registers specified by `PDSC_RPD_IMASK` and `PDSC_RDP_FMASK` in the register save area based on `PDSC_RPD_RSA_OFFSET`.
   – Store the return address in the register save area.
4. For a register frame procedure (`PDSC_FLAGS_REGISTER_FRAME` is 1), copy the return address to the register specified by `PDSC_RPD_SAVE_RA` if the value is not already there.
5. Execute `trapb`, if required. (See Section 5.1.12 for details.)

6. For a variable-size stack frame procedure (PDSC_FLAGS_BASE_REG_IS_FP is 1), copy the SP value to register FP.

   This step must be completed in exactly one instruction that modifies the FP and that instruction must be the last instruction in the prologue.

When these steps have been completed, the executing procedure is said to become *current* for the purposes of exception handling. The handler for a procedure will not be called except when that procedure is current.

The remainder of this section contains the following:

- A description of prologue length
- A description of frame pointer conventions
- An example of entry code for a stack frame
- An example of entry code for a register frame

3.2.6.1.1 Prologue Length

As a general rule, it is valid to include instructions in the prologue (in addition to those that are required) in order to take advantage of available processor cycles that are not otherwise used. However, any such additional instructions must not cause an exception that would need to be handled by that procedure if that exception would be raised after the procedure became current. (An exception is considered to not be handled by a procedure if it is known that the handler will always resignal that exception.)

3.2.6.1.2 Frame Pointer Conventions

After the procedure prologue is completed, the register indicated by PDSC_FLAGS_BASE_REG_IS_FP must contain the frame pointer of the stack frame. The **frame pointer** is the address of the lowest-addressed byte of the fixed portion of the stack frame allocated by the procedure prologue. The value of the frame pointer is the value of PDSC_RPD_FRAME_SIZE subtracted from the value of the stack pointer upon procedure entry.

For fixed frame procedures, the frame pointer is the stack pointer. In these cases, the stack pointer is not modified by that procedure after the instruction in that procedure prologue specified by PDSC_RPD_SP_SET.

3.2.6.1.3 Entry Code Example for a Stack Frame Procedure

This section contains an entry code example for a stack frame procedure. The example assumes the following:

- Registers \$9 – \$11 and \$f2 – \$f3 are saved and restored.
- PDSC_RPD_RSA_OFFSET is equal to 16 bytes.

- The procedure has a static exception handler that does not re-raise arithmetic traps.
- The procedure uses a fixed amount of stack (PDSC_FLAGS_BASE_REG_IS_FP is 0).

The following example illustrates a Stack Frame Procedure.

```
ldah   GP, off_hi($27)    #Compute the correct GP value.
lda    GP, off_lo(GP)
lda    SP,-SIZE(SP)       #Allocate space for a new stack frame.
stq    $26,16(SP)         #Save the return address.
stq    $9,24(SP)          #Save the first integer register.
stq    $10,32(SP)         #Save the next integer register.
stq    $11,40(SP)         #Save the next integer register.
stt    $f2,48(SP)         #Save the first floating-point register.
stt    $f3,56(SP)         #Save the last floating-point register.
trapb                     #Force any pending hardware exceptions
                          #   to be raised.
#The called procedure is now the current procedure.
```

3.2.6.1.4 Entry Code Example for a Register Frame Procedure

This section contains an entry code example for a register frame procedure. The example assumes the following:

- The called procedure has no static exception handler.
- PDSC_RPD_SAVE_RA and PDSC_RPD_ENTRY_RA specify $26.
- The procedure utilizes a fixed amount of stack storage (PDSC_FLAGS_BASE_REG_IS_FP is 0).

The following example illustrates a Register Frame Procedure.

```
ldah   GP, off_hi($27)     #Compute the correct GP value.
lda    GP, off_lo(GP)
lda    SP,-SIZE(SP)        #Allocate space for a new stack frame.
#The called procedure is now the current procedure.
```

3.2.6.2 Exit Code Sequence

The end of procedure entry code can be determined easily by using a PC value together with the PDSC_RPD_ENTRY_LENGTH value. Because there can be multiple return points from a procedure, detecting that a procedure exit sequence is being executed is not as straightforward. Unwind support routines must be able to detect if the stack pointer has been reset and if not, know how to reset it. The exit sequence can be detected by requiring a reserved instruction sequence.

The next sections provide the following information:

- A discussion of the reserved instruction sequence
- The steps involved in an exit code sequence
- An example of exit code for a stack frame

- An example of exit code for a register frame

3.2.6.2.1 Reserved Instruction Sequence for a Procedure Exit

To allow the stack to be properly restored during an unwind, a reserved instruction or sequence of instructions must be used. None of these sequences can be used in any other way.

The following reserved instruction must appear at every exit point from any procedure that uses the stack (`PDSC_RPD_FRAME_SIZE` is not 0):

```
ret  $31,($n),0001   #Return to the caller with usage hint 0001
```

Note

The term **usage hint**, shown in the comment in the previous example, refers to the value of the branch prediction bits encoded in the `ret` instruction. The section on control instructions in the *Alpha Architecture Reference Manual* documents that these bits, <13:0> of the instruction longword, are reserved to software when the instruction is `ret` or `jsr_coroutine`.

This calling standard further requires that these bits contain the value 0001 (hex) for procedure returns and 0000 otherwise. The occurrence of the usage hint value 0001 identifies a `ret` instruction as one that is reserved for use only as described in the *Alpha Architecture Reference Manual*. The `ret` instructions can be used for other purposes, provided they contain a usage hint value of 0000. Those `ret` instructions will not be recognized and treated in a special way for the purposes of exception handling or unwinding.

In almost all cases, the return address register ($n) used will be $26, because it is required to be reloaded prior to the procedure return.

For any such procedure that does not return a value on the stack, the `ret` instruction must be immediately preceded by either of the two reserved stack resetting instructions. The following examples show the two different reserved stack-resetting instructions:

```
lda    SP,*               #Reset the stack.
ret    $31,(*),0001       #Return to the caller with usage hint.

addq   *,*,SP             #Reset the stack.
ret    $31,(*),0001       #Return to the caller with usage hint.
```

Thus, any `lda` instruction whose destination is the SP register or any `addq` instruction whose destination is the SP register is interpreted as part of a procedure exit sequence when it is immediately followed by the reserved procedure return instruction.

A stack-resetting instruction might not be present in the case of a procedure that returns a result on the top of the stack. However, if such an instruction is present, it will immediately preced the reserved `ret` instruction.

Furthermore, for any such procedure that has `PDSC_FLAGS_BASE_REG_IS_FP` set to 1, the resulting sequence must immediately follow the FP reloading instruction as in the following example:

```
ldq    FP,*                #Restore the FP.
lda    SP,*                #Or addq *,*,SP to reset the stack.
ret    $31,(*),0001        #Return to the caller with usage hint.
```

Thus, any `ldq` instruction whose destination is the FP register ($15) is interpreted as part of a procedure exit sequence when it is immediately followed by the reserved procedure return instruction or by a stack resetting instruction that is in turn immediately followed by the reserved procedure return instruction.

Procedures that do not use the stack do not need to use these reserved instruction sequences.

The unwind support code uses the exit code sequences to make the following assumptions about an interrupted PC value:

- If the PC points within the prologue, the registers still have their original contents. Only SP must be reset if the PC is beyond `PDSC_RPD_SP_SET`. Then the unwind can proceed.
- If the PC points at a `ret $31,(*),0001` instruction, SP has already been reset and the registers have already been restored, so the unwind can proceed.
- If the PC points to an `lda SP,*` (or an `addq *,*,SP)` instruction that is immediately followed by the instruction described previously, the registers have already been restored. But SP must be incremented by `PDSC_RPD_FRAME_SIZE` before the unwind can proceed.
- If the PC points to an `ldq FP,*` instruction that is immediately followed by either of the instructions described previously, `PDSC_FLAGS_REGISTER_FRAME` is 0 and `PDSC_FLAGS_BASE_REG_IS_FP` is 1, all registers other than FP have been restored. FP still retains the frame base pointer, which should be copied to SP. Then FP must be restored and SP incremented by `PDSC_RPD_FRAME_SIZE` for the unwind to proceed.
- In all other cases, the registers must be restored and SP reset for the unwind to proceed.

When a procedure has executed the first instruction of one of these reserved sequences, the procedure becomes no longer *current* for the purposes of exception handling. The handler for a procedure will not be called in the midst of one of these reserved instruction sequences within that procedure.

3.2.6.2.2 Exit Code Sequence Steps

When a procedure returns, the exit code must restore the caller's context, synchronize any pending hardware exceptions, and make the calling procedure current by returning control to it. The following list contains the exit code sequence steps. The program performs step 1, followed by steps 2 through 5 in any order, followed by steps 6 through 8 in exact order.

1. If the GP register has been modified or a call has been made, restore the GP register to the GOT segment pointer of the current procedure.
2. For a variable-size stack frame procedure that does not return a value on the top of stack (`PDSC_FLAGS_BASE_REG_IS_FP` is 1), copy FP to SP.
3. For a stack frame procedure (`PDSC_FLAGS_REGISTER_FRAME` is 0), reload any saved registers from the register save area as specified by `PDSC_RPD_RSA_OFFSET`. Note that, for a variable-size stack frame procedure (`PDSC_FLAGS_BASE_REG_IS_FP` is 1), FP is not reloaded in this step. For a fixed-size stack frame procedure (`PDSC_FLAGS_BASE_REG_IS_FP` is 0), $15 is reloaded if it was saved on entry.
4. Reload the register that held the return address on entry with the saved return address, if necessary.

   For a stack frame procedure (`PDSC_FLAGS_REGISTER_FRAME` is 0), load the register designated by `PDSC_RPD_ENTRY_RA` ($26 in a standard call) with the return address from the register save area as specified by `PDSC_RPD_RSA_OFFSET`.

   For a register frame procedure (`PDSC_FLAGS_REGISTER_FRAME` is 1), copy the return address from the register specified by `PDSC_RPD_SAVE_RA` to the register designated by `PDSC_RPD_ENTRY_RA`.
5. Execute `trapb`, if required. (See Section 5.1.12 for details.)
6. For a variable-size stack frame procedure (`PDSC_FLAGS_REGIS-TER_FRAME` is 0 and `PDSC_FLAGS_BASE_REG_IS_FP` is 1), reload $15 (FP) as would be done for any other saved register.

   After any necessary calculations, this step must be completed by exactly one instruction, as described in Section 3.2.6.1.
7. If a function value is not being returned on the stack, restore SP to the value it had at procedure entry by adding the value in

PDSC_RDP_FRAME_SIZE to SP. In some cases, the returning procedure leaves SP pointing to a lower stack address than it had on entry to the procedure, as specified in Section 4.1.7.

   After any necessary calculations, this step must be completed by exactly one instruction, as described in Section 3.2.6.1.

8. Execute the `ret $31,($n),0001` instruction, as described in Section 3.2.6.2.1, to return control to the calling procedure. In almost all cases the $n used will be $26 (the return address register) because its value must be restored before the call returns.

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

3.2.6.2.3 Exit Code Example for a Stack Frame Procedure

The following example shows the return code sequence for the stack frame procedure example in Section 3.2.6.1.3. This code fragment assumes that the computed GP value was saved in the preserved register $11:

```
mov    $11,GP         #Restore this routine’s GP value.
ldq    $26,16(SP)     #Get the return address.
ldq    $9,24(SP)      #Restore the first integer register.
ldq    $10,32(SP)     #Restore the next integer register.
ldq    $11,40(SP)     #Restore the next integer register.
ldt    $f2,48(SP)     #Restore the first floating-point register.
ldt    $f3,56(SP)     #Restore the last floating-point register.
trapb                 #Force any pending hardware exceptions to be
                      #   raised.
lda    SP,SIZE(SP)    #Restore the SP.
ret    $31,($26),0001 #Return to the caller with the usage hint.
```

3.2.6.2.4 Exit Code Example for a Register Frame Procedure

The following example shows the return code sequence for the register frame procedure example in Section 3.2.6.1.4.

```
lda    SP,SIZE(SP)    #Restore the SP.
ret    $31,($26),0001 #Return to the caller with the usage hint.
```

4 Data Manipulation

This chapter discusses the passing and storage of data. The topics included are:

- Data passing (Section 4.1)
- Data allocation (Section 4.2)

4.1 Data Passing

The following sections define the calling standard conventions for passing data between procedures in a call chain. An **argument item** represents one unit of data being passed between procedures. The following topics are covered:

- Mechanisms for passing argument items (Section 4.1.1)
- Normal argument list structures (Section 4.1.2)
- Homed memory argument list structures (Section 4.1.3)
- Argument lists and high-level languages (Section 4.1.4)
- Unused bits in passed data (Section 4.1.5)
- Sending data (Section 4.1.6)
- Returning data (Section 4.1.7)

4.1.1 Argument Passing Mechanisms

This Tru64 UNIX calling standard defines three classes of argument items according to the mechanism used to pass the argument:

- 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 a parameter.

- Reference

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

  Note that this standard does not define a standard set of descriptors. Consequently, descriptors cannot be used as part of a standard call.

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 of the three mechanisms must be used by a given language compiler. Language semantics and interoperability considerations might require different mechanisms to be used in different situations.

4.1.2 Normal Argument List Structure

The **argument list** in a Tru64 UNIX 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 represented in 64 bits.

An 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 fashion that can produce a physically discontiguous argument list. Registers $16 – $21 and $f16 – $f21 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 4–1 specifies the standard locations in which argument items can be passed.

Table 4–1: Argument Item Locations

Argument Item	Integer Registers	Floating-point Registers	Stack
1	$16	$f16	-
2	$17	$f17	-
3	$18	$f18	-
4	$19	$f19	-

Table 4–1: Argument Item Locations (cont.)

Argument Item	Integer Registers	Floating-point Registers	Stack
5	$20	$f20	-
6	$21	$f21	-
7 ... `n`			`0(SP) ...  (n-7)*8(SP)`

The following general rules determine the location of any specific argument:

- All argument items are passed in the integer registers or on the stack, except argument items that have 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 in any row in Table 4–1 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, argument item 3 is assigned to $18 and argument item 4 is assigned to $f19.
- A single- or double-precision complex value passed by immediate value is treated as two arguments for the purpose of this standard, with the real part coming first. For example, if the real part of a complex value is passed as the sixth argument item in register $f21, the imaginary part will be passed in memory as the seventh argument item.

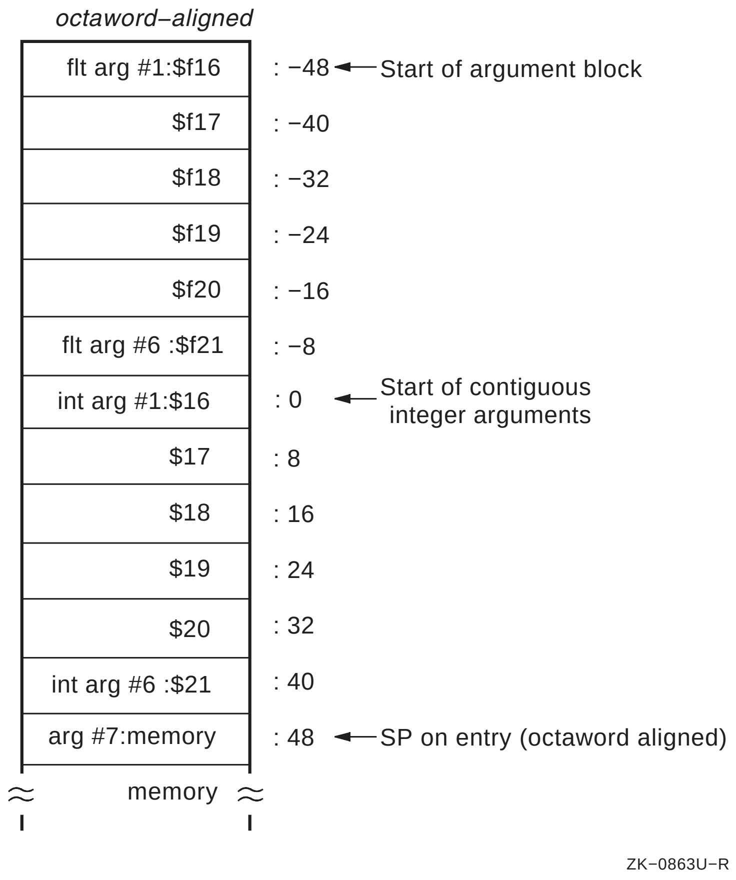
  An extended-precision complex value is passed by reference using a single integer or stack argument item. (An extended-precision complex value is not passed by immediate value because the component extended-precision floating values are not passed by value. See also Section 4.1.6.)

The argument list, including the in-memory portion, as well as 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.

4.1.3 Homed Memory Argument List Structure

It is, in certain cases, useful to form a contiguous in-memory structure that includes the contents of all the formal parameter values in the program; for example, C procedures that use varying length argument lists). In nearly all these cases, a compiler can arrange to allocate and initialize this structure so that those parameter values passed in registers are placed adjacent to those parameters passed on the stack, without making a copy of the stack arguments. The storage for the parameters passed in registers is called the **argument home area**. (See Figure 3–1 and Figure 3–2.) Figure 4–1 shows the resulting in-memory homed argument list structure.

Figure 4–1: In-Memory Homed Argument List Structure



Generally, it is not possible to tell statically whether a particular argument is an integer or floating-point argument. Therefore, it is necessary to store integer and floating-point register argument contents in this structure. However, it is sometimes possible to determine statically that there are no floating-point arguments anywhere either in registers or on the stack. In this case, the first six entries can be omitted. To facilitate this special case, the address used to reference this structure is always the address of the first integer argument position.

The C-language type `va_list` is used to iterate through a variable argument list. The `va_list` type can be defined as follows:

```
typedef struct {
    char     *base;
    int      offset;
    } va_list;
```

To load the next integer argument, the program reads the quadword at location (base+offset) and adds 8 to offset. To load the next floating-point argument, if offset is less than or equal to 6*8, the program reads the quadword location (base+offset–6*8). Otherwise, the program reads the quadword at location (base+offset). In both cases, the program adds 8 to offset. For details, see the file /usr/include/stdarg.h.

4.1.4 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. The $16 or $f16 register is allocated to the first argument; 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 Tru64 UNIX calling standard argument items.
- Each argument item has 64 bits.
- A null or omitted argument, for example CALL SUB(A,,B), is represented by an argument item containing 0.

  Arguments passed by immediate value cannot be omitted unless a default value is supplied by the language. (This restriction makes it possible for 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 those for embedded null or omitted arguments.

4.1.5 Unused Bits in Passed Data

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

Table 4–2 defines the various data type requirements for size and their extension to set or clear unused bits.

Table 4–2: Unused Bits in Passed Data

Data Type	Type Designator (bytes)	Data Size Type	Register Extension Type	Memory Extension
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
IEEE floating single S	FS	4	Hard	Data32
IEEE floating double T	FT	8	Hard	Data64
IEEE floating extended X	FX	16	n/a	n/a
IEEE floating single S complex	FSC	2*4	2*Hard	2*Data32
IEEE floating double T complex	FTC	2*8	2*Hard	2*Data64
IEEE floating extended X complex	FXC	2*16	n/a	n/a
Structures	N/A		Nostd	Nostd
Small arrays of 8 bytes or less	N/A	≤ 8	Nostd	Nostd
32-bit address	N/A	4	Sign64	Sign64
64-bit address	N/A	8	Data64	Data64

The following table contains the definitions for the extension type symbols used in Table 4–2:

Sign Extension Type	Definition
`Sign32`	Sign-extended to 32 bits. The state of bits <63:32> is *unpredictable*.
`Sign64`	Sign-extended to 64 bits.
`Zero32`	Zero-extended to 32 bits. The state of bits <63:32> is *unpredictable*.
`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 with 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 with each handled as `Data64`.
`Hard`	Passed in the layout defined by the *Alpha Architecture Reference Manual*.
`2 * Hard`	Two double-precision parts of the complex value are stored in a pair of registers as independent floating-point values with each handled as `Hard`.
`Nostd`	The state of all high-order bits not occupied by the data is *unpredictable* across a call or return.

Note

`Sign64`, when applied to a longword logical, duplicates bit 31 through bits <63:32>. This duplication can cause the 64-bit integer value to appear negative. However, careful use of 32-bit arithmetic and 64-bit logical instructions (with no right shifts) will preserve the 32-bit unsigned nature of the argument.

Because of the varied rules for sign extension of data when passed as arguments, 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.

4.1.6 Sending Data

The following sections define the calling standard requirements for mechanisms to send data and the order of argument evaluation.

4.1.6.1 Sending Mechanism

In Section 4.1.1, the allowable argument-passing mechanisms are immediate value, reference, and descriptor. The following list describes the requirements for using these mechanisms.

- 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 less than or equal to 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 less than or equal to 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 data type, to fill all bits of each argument list item as specified in Table 4–2. (Note that unoccupied bits may be left undefined for records and single-precision floating point according to that table.).

  A single- or double-precision complex value is passed as two single- or double-precision floating-point values, respectively.

  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 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 used for passing records. Thus, for example, a 26-byte string can be passed by value in four integer registers.

- By reference – nonparametric

  Nonparametric arguments (that is, arguments for which associated information such as string size and array bounds is not required) may be passed by reference in a standard call. This includes extended-precision floating and extended-precision complex values.

- By reference – parametric

  Parametric arguments (that is, arguments for which associated information such as string size and array bounds must be passed to the caller) may be passed by reference followed by one or more immediate arguments for the parametric values. (The parametric values do not need to immediately follow the reference arguments to which they apply.)

  Note

  This standard does not define interlanguage conventions for calls with parametric arguments.

- By descriptor

  Parametric arguments (that is, arguments for which associated information such as string size and array bounds must be passed to the caller) can be passed by a single descriptor.

  Note

  This standard does not define a standard set of descriptors for interlanguage use.

Note that extended–precision floating-point values are not passed using the immediate value mechanism. Instead, they are passed using the by-reference mechanism. (When by-value semantics are required, however, it might be necessary to make a copy of the actual parameter and pass a reference to that copy to avoid improper alias effects.)

Note also that when a record is passed by immediate value, the component types have no bearing on how the argument is aligned. The record will always be quadword-aligned.

4.1.6.2 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. 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.

4.1.7 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. 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.

The following sections describe each of the three standard mechanisms for returning function values.

4.1.7.1 Function Value Return by Immediate Value

The following list describes the two types of immediate value function returned:

- Nonfloating-point function value return by immediate value

  A function value is returned by immediate value in register $0 *if and only if* the type of function value is one of the following:

  – Nonfloating-point scalar data type with size known (at compile time) to be less than or equal to 64 bits

  – Set, implemented as a bit vector, with size known (at compile time) to be less than or equal to 64 bits

  No form of string, record, or array can be returned by immediate value. Two separate 32-bit entities cannot be returned in $0.

  A function value of less than 64 bits returned in $0 must have its unoccupied bits extended (as appropriate, depending on the data type) to a full quadword. (See Table 4–2 for details.)

- Floating-point function value return by immediate value

  A function value is returned by immediate value in register $f0 *if and 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 *if and only if* it is a complex single- or double-precision floating-point value (complex F, D, G, S, or T). The real part is in $f0 and the imaginary part is in $f1.

  Note that extended–precision floating point and extended–precision complex values are returned by reference as described in the following section.

4.1.7.2 Function Value Return by Reference

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

- Its size is known to the calling procedure and the called procedure, but the value cannot be returned by immediate value because, for example, the function value requires more than 64 bits or the data type is a string, record, 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 address of the function value.

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 that is 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.

4.1.7.3 Function Value Return by Descriptor

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

- It cannot be returned by immediate value because, for example, the function value requires more than 64 bits or the data type is a string, record, or an array type.
- Its size is not known to the calling procedure or the called procedure.
- It can be returned in a contiguous region of storage.

Function results returned by descriptor are not permitted in a standard call.

Typically, the called routine creates the return object on its stack and leaves it there on return. This process is referred to as the **stack return** mechanism. The exit code of the called routine does not restore SP to its value before the call because, if it did, the return value would be left unprotected in memory below SP. The calling routine must be prepared for SP to have a different value after the call than the pointer had before the call.

4.2 Data Allocation

Data allocation refers to the method of storing data in memory. The following sections cover these topics:

- Data alignment (Section 4.2.1)
- Granularity of memory access (Section 4.2.2)
- Record layout conventions (Section 4.2.3)

4.2.1 Data Alignment

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

To avoid such performance degradation, all data values on Alpha systems should be naturally aligned. Moreover, the base address of a global variable or aggregate (but not necessarily the components of the aggregate) that is shared across a standard call must be quadword aligned. Table 4–3 shows the data alignment requirements for non-global data.

Table 4–3: 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)
Single-precision real value	Address that is a multiple of 4 (longword alignment)
Double-precision real value	Address that is a multiple of 8 (quadword alignment)
Extended-precision real value	Address that is a multiple of 16 (octaword alignment)
Single-precision complex value	Address that is a multiple of 4 (longword alignment)
Double-precision complex value	Address that is a multiple of 8 (quadword alignment)
Extended-precision complex value	Address that is a multiple of 16 (octaword alignment)
Data types larger than 64 bits	Quadword or greater alignment. (Alignments larger than quadword are language-specific or application defined)

For aggregates such as strings, arrays, and records, the data type to be considered for purposes of alignment is not the aggregate itself, but the elements that make up the aggregate. The alignment requirement of an aggregate is that all elements of the aggregate be naturally aligned. Varying 8-bit character strings, for example, 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, regardless of the extent of the array.

Note that the rules in Section 4.1.6.1 for passing by value an argument that is a record always provide quadword alignment of the record value independent of the normal alignment requirement of the record. If deemed appropriate by the implementation, normal alignment can be established within the called procedure by making a copy of the record argument at a suitably aligned location.

4.2.2 Granularity of Memory

On the Alpha architecture, memory is byte addressable and the smallest unit in which memory may be accessed — the **granularity** — is one byte. However, the original Alpha architecture provided only longword granularity; as a result, longword granularity is supported by many existing Alpha processors.

Even for longword-sized data, it is often expedient for execution efficiency to access memory in quadword units. In the presence of multiple threads of execution (whether on multiple processors or a single processor), allocation of more than one data element within a single quadword can lead to more complicated access sequences (for example, using `ldx_l/stx_c`) and/or latent and hard to diagnose errors because of nonobvious and implicit data sharing. Therefore, it is generally recommended that independent variables (that is, variables not combined in a larger aggregate) be allocated on quadword boundaries.

4.2.3 Record Layout Conventions

The Tru64 UNIX calling standard record layout conventions are designed to provide good run-time performance on all implementations of the Alpha architecture. Only the standard record layouts may be used across standard interfaces or between languages. Languages can support other language-specific record layout conventions, but such other record layouts are nonstandard.

The aligned record layout conventions ensure the following:

- 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 might be embedded.
- Layout and alignment of a subrecord are the same as if that data item was 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 Tru64 UNIX environments and other environments.

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; that is, it must be 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, which is a byte, word, longword, or quadword in size, and by 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. If the resulting allocation crosses an alignment boundary, the field is allocated at the next byte boundary that is aligned as required for the underlying type. (In this latter case, the space skipped over is left permanently unallocated.)

  In addition, the alignment of the record as a whole is increased to that of the underlying integer type, if necessary.
- 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 requirement also holds 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 designation leaves unused space at its end. The length of the complete array is the sum of the lengths of its elements.

5
Event Processing

This chapter discusses specifications related to events that are outside the normal program flow. The topics covered are:

- Exception handling (Section 5.1)
- Unwinding (Section 5.2)

5.1 Exception Handling

This section discusses the following considerations involved in the notification and handling of exceptional events during the course of normal program execution. An exception is a condition in the current software and/or hardware state that should be noted or fixed. The exception handler deals with the exception condition.

The next sections cover the following topics:

- Requirements for exception handling (Section 5.1.1)
- An overview of exception handling (Section 5.1.2)
- The kinds of exceptions (Section 5.1.3)
- Status values and exception codes (Section 5.1.4)
- Exception records (Section 5.1.5)
- Frame-based exception handlers (Section 5.1.6)
- How handlers become established (Section 5.1.7)
- How exceptions are raised (Section 5.1.8)
- The search for and invocation of exception handlers (Section 5.1.9)
- Modification of exception records and context by handlers (Section 5.1.10)
- Handler completion and return value (Section 5.1.11)
- Exception synchronization and continuation from exceptions (Section 5.1.12)
- Coexistence of exception and signal handling (Section 5.1.13)

5.1.1 Exception Handling Requirements

This Tru64 UNIX calling standard supports the following exception handling capabilities:

- Reliable programmer and program control over response to exceptions and reporting of such exceptions, as well as over the flow of control when exceptions occur
- Orderly termination of layered applications
- Correct and predictable exception handling in a multilanguage environment
- Construction of modular, maintainable multilanguage applications
- Parallel multithreaded application execution, including:
  - Per-thread exception handling
  - Handling of asynchronous exceptions
  - Safe thread exit in a multithreaded environment
- Coexistence and interoperation with POSIX-defined signal handling

5.1.2 Exception Handling Overview

When an exception occurs (is raised), the following events take place:

- The normal flow of control in the current thread is interrupted.
- The current context is saved.
- Control is transferred to the exception-handling support code.
- The exception-handling support code collects the exception information and then enters a section of the support code called the **exception dispatcher**.
- The exception dispatcher searches for exception handlers and invokes them in the proper sequence.

When a handler is invoked, it is called as a procedure with arguments that describe the following:

- The nature of the exception
- The environment within which the exception was raised
- The environment within which the handler was established

When the handler is called, the exception is said to be delivered to the handler.

The handler can respond to the exception in several ways, including various combinations of the following:

- Perform some action that affects the context of the thread (such as correcting the circumstances that led to the exception being raised).
- Modify or augment the description of the exception.
- Raise a **nested exception**, causing another exception to occur in the context of the exception handler or in a procedure called directly or indirectly by the handler.

When an exception handler has finished processing an exception, it must indicate this state in one of the following ways:

- Reraise the exception.

  The handler indicates that the exception handling support code should reraise the exception and resume the search for a new handler.
- Continue the normal program flow.

  The handler indicates that the exception handling support code should continue execution of the interrupted thread at the location indicated by the saved exception program counter.
- Unwind from the current operation.

  The handler performs an unwind operation that causes the exception handling support code to resume execution of the thread at a point other than the point at which it was interrupted or to terminate the execution of the thread.

All exceptions are handled using the same interfaces, data structures, and algorithms. That is, exception handling is unified for all kinds of exceptions, regardless of their origins. The interfaces and data structures are defined in the file `/usr/include/excpt.h`.

Each exception has an **exception value** that identifies the type of exception, such as subscript range violation or memory access control violation. Exceptions can also be associated with one or more **exception qualifiers** (such as the name of an array and the subscript that was out of range, or an address associated with a memory access control violation).

5.1.3 Kinds of Exceptions

There are three kinds of exceptions:

- General exceptions – those caused by general software or hardware notification mechanisms
- Unwind exceptions – those caused by unwind operations
- Signal exceptions – those caused in support of the POSIX 1003.1 `signal()` routine

5.1.3.1 General Exceptions

General exceptions are divided into two categories:

- Software-caused

  A software-caused general exception is raised when an exception-raising procedure is invoked. This type of exception is always delivered to the thread that made the call.

  A software-caused exception can be raised at any point during thread execution. Applications and language run-time libraries can raise general exceptions to notify a thread of some exceptional (noteworthy) state in the current thread context. For example, subscript range-checking failures and assertion-checking failures can be raised as software-caused general exceptions.

- Hardware-caused

  A hardware-caused general exception occurs when a thread performs some action that causes an exceptional state to exist in the hardware. Such a state will cause the currently active thread to be interrupted. A hardware-caused general exception is always delivered to the thread that executed the instruction that caused the exception.

  The following characteristics are specific to individual hardware exceptions:

  – Exactly which hardware events can result in exceptions

  – The state of the machine when a hardware exception occurs

  – The interpretation of the exception-related information that is delivered to a user mode thread

  – The circumstances under which execution can be continued

  Hardware exceptions are fully defined in the *Alpha Architecture Reference Manual*.

  In the Tru64 UNIX for Alpha systems environment, hardware exceptions that are not handled by the operating system itself are reported to user level in the form of a POSIX signal. Such signals are then further interpreted as exceptions and are handled as described in this calling standard. See Section 5.1.13 for details on how this signal interpretation is achieved.

5.1.3.2 Unwind Exceptions

Unwind exceptions result from the invocation of the unwind support code by a thread. These exceptions are always delivered to the thread that invoked the unwind.

Unwind exceptions are delivered as part of the notification process that an unwind is in progress. (See Section 5.2 for details.)

5.1.3.3 Signal Exceptions

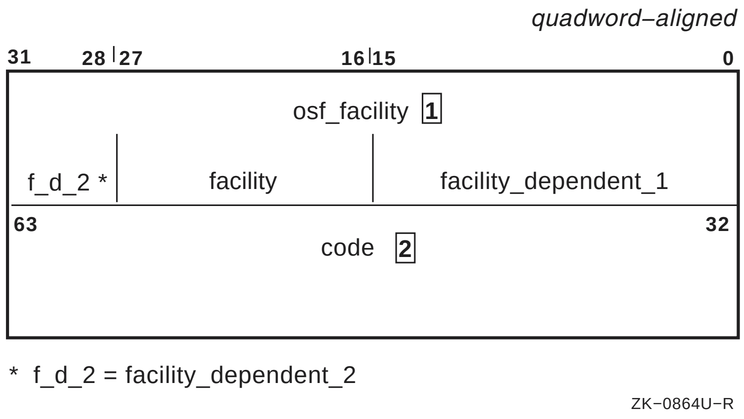
Signal exceptions result from the delivery of a POSIX signal. This signal is subsequently converted into an exception that can be handled using the capabilities defined by this calling standard.

5.1.4 Status Values and Exception Codes

A **status value** is a quadword that can be used as a return value from a procedure call to indicate success, failure, or other information about the requested operation. A status value can also be used as an exception code to indicate the reason that an exception is being raised.

Figure 5–1 shows the components of a status value structure.

Figure 5–1: Status Value Representation



The components of this representation are as follows:

1 osf_facility

A facility code indicating the software component that defines this status value. Only those values defined in the file `/usr/include/excpt.h` can be used. This component consists of three subfields:

– `facility`

This field has the value ffe(hex) on Tru64 UNIX for Alpha systems to distinguish the status value from those of other operating systems.

– `facility_dependent_1`

This field indicates the particular programming language or system associated with the value.

  – `facility_dependent_2`

    This field serves the same routine as `facility_dependent_1`.

2 *code*

  This field contains a value for a particular status condition.

There are several ways for a software system or application to define its own status values:

- It may already have status values from another operating system, such as OpenVMS.
- It can register values for the fields `facility_dependent_1` and `facility_dependent_2` with HP. The `facility` field will have the value ffe(hex). All values of the `code` field will be available.
- It can use the predefined `osf_facility` value `EXC_C_USER` (ffe0009(hex)). However, these status values might conflict with status values of other operating systems.

5.1.5 Exception Records

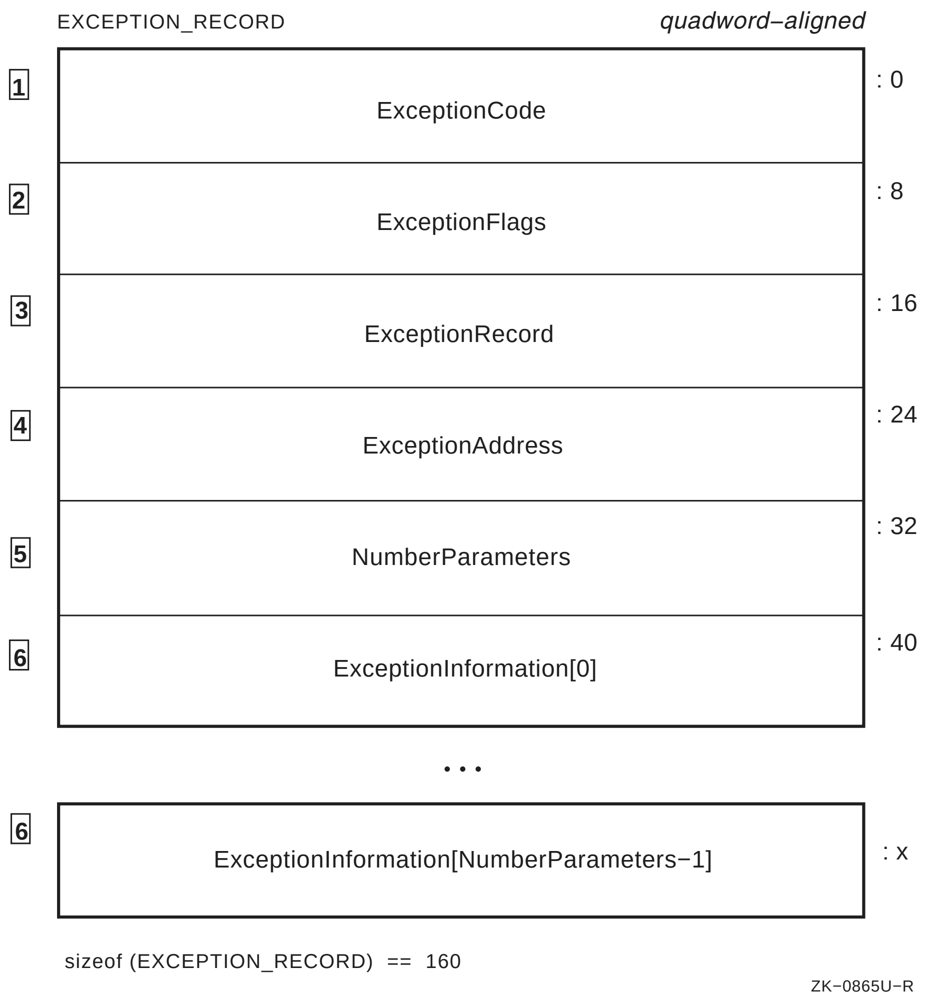
The fundamental data structure for describing exceptions is the **exception record**. Exception records can be joined together by handlers to form a linked list. Each record in a list describes one exception.

The first exception record in the list describes the **primary exception**. **Secondary exceptions** can be specified by adding exception records to the list. Secondary exceptions qualify or elaborate the primary exception. In some cases, they are raised at the same time as the primary exception. In other cases, a handler can add new secondary exceptions to the list before handling or reraising the exception.

Storage for exception records can be allocated in read-only memory. The exception record that is passed to a handler is a separate read-write copy constructed from information in the original exception record augmented with additional information.

Figure 5–2 shows the format of an exception record.

Figure 5–2: Exception Record Format



$\boxed{1}$ `ExceptionCode` is an exception code value. (See Section 5.1.4.)

$\boxed{2}$ `ExceptionFlags` is a bit field of flags that further qualify the exception. These flag bits are significant only in the primary exception record; their state is *unpredictable* in secondary exception records. `ExceptionFlags` bits are logically divided into two groups: **detail flag** and **environment flags**. Detail flags provide additional information about the exception. Environment flags provide additional information about the environment in which the exception is being delivered.

The following `ExceptionFlags` bits are detail flags that give additional details:

– `EXCEPTION_NONCONTINUABLE` (bit 0)

If EXCEPTION_NONCONTINUABLE is 1, an exception handler must not return ExceptionContinueExecution.

– EXCEPTION_EXIT_UNWIND (bit 2)

  If EXCEPTION_EXIT_UNWIND is 1, the exception handler is being invoked because of an unwind operation that will terminate execution of the thread.

– EXCEPTION_UNWINDING (bit 1)

  If EXCEPTION_UNWINDING is 1, the exception handler is being invoked because of a general unwind operation with the semantics of longjmp().

The following ExceptionFlags bits are environment flags that give additional information about the environment at the time of exception delivery:

– EXCEPTION_NESTED_CALL (bit 4)

  If EXCEPTION_NESTED_CALL is 1, an exception or unwind is in progress at the time this exception is delivered.

– EXCEPTION_STACK_INVALID (bit 3)

  If EXCEPTION_STACK_INVALID is 1, the stack is invalid.

  Note that this flag is for use by system software; it will never be 1 in an exception record delivered to a normal handler.

– EXCEPTION_TARGET_UNWIND (bit 5)

  If EXCEPTION_TARGET_UNWIND is 1, this frame is the target frame of an unwind operation. (This flag can be useful in allowing a language-specific handler to perform proper handling for the last step of an unwind.)

– EXCEPTION_COLLIDED_UNWIND (bit 6)

  If EXCEPTION_COLLIDED_UNWIND is 1, an unwind collision has occurred. (See Section 5.2.6 for information on multiply active unwind operations).

All FLAGS bits other than those defined in the preceding lists must be zero.

3 ExceptionRecord is zero or contains the address of the next exception record in the list.

4 ExceptionAddress is the address of the instruction causing the exception.

For a hardware, signal, or asynchronous software exception, this field contains the address of the instruction at which the hardware, signal, or asynchronous exception interrupted execution of the thread.

For a synchronous software exception, this field contains the address of the call instruction that invoked the library routine for raising the exception.

This field is significant only in primary exception records; its contents are *unpredictable* in secondary exception records.

5 `NumberParameters` is the number of exception-specific qualifiers in the exception record.

6 Each `ExceptionInformation` [*n*] value is a single quadword that provides additional information specific to the exception. The quadword can also contain information intended for display in messages.

5.1.5.1 Exception Records for General Exceptions

In the case of software-caused exceptions, the information in the exception records for general and unwind exceptions can vary widely from a simple single-exception value to a long chain of exceptions and exception qualifiers. This calling standard defines the conventions for constructing these exception records. However, a complete enumeration of all possible combinations is beyond the scope of this document.

All Alpha hardware exceptions have exception information associated with them. This information can be as little as the exception type and exception PC or as much as three registers worth of additional information. The specific information that is supplied with each exception type is defined in the *Alpha Architecture Reference Manual*.

Hardware exceptions are reported to user mode in the form of POSIX signals. See Section 5.1.5.3.

5.1.5.2 Exception Records for Unwind Exceptions

Unwind exceptions have at least one of the following flags set to 1:

- `EXCEPTION_UNWINDING`
- `EXCEPTION_EXIT_UNWIND`
- `EXCEPTION_TARGET_UNWIND`

The `ExceptionCode` for unwind exceptions contains the reason code for the unwind along with any supplied qualifiers. This information is similar to the contents of the `ExceptionCode` field for general exceptions.

5.1.5.3 Exception Records for Signal Exceptions

In exception records for signal exceptions, the value of the `ExceptionCode` field includes the signal number. Any additional exception qualifiers that are present can further qualify the signal. These qualifiers are most useful for hardware-generated signals.

For example, in a POSIX-conforming environment, all arithmetic exceptions are delivered with a signal number `SIGFPE`. The qualifiers can indicate whether the signal was caused by a floating underflow, integer overflow, or other arithmetic exception. (In a POSIX-conformant environment, no mechanism is provided for a user program to deliver a signal with a qualifier of any kind. Thus, the presence of such a qualifier occurs because it was produced by some system facility, such as the hardware exception dispatching code.) See Section 5.1.13 for a discussion of exception and signal handling coexistence.

5.1.6 Frame-Based Exception Handlers

A **frame-based exception handler** is established when a procedure whose descriptor specifies an exception handler becomes current. Thus, frame-based handlers are usually associated with a procedure at compile time and are located at run time through the procedure descriptor. These exception handlers are normally used to implement a particular language's exception handling semantics.

The frame-based exception handlers that can be invoked are those established by active procedures, from the most current procedure to the oldest predecessor.

An exception handler that conforms to this calling standard should not handle any exception that its establisher did not cause, unless there is a prior agreement between the writers of the exception handler and the writers of the code that raised the exception.

Exceptions can be raised and unwind operations that cause exception handlers to be called can occur when the current value of at least one variable is in a register rather than in memory. Therefore, 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 requirement can be violated for specific memory locations only by agreement between the handler and all procedures that might access those memory locations. The effects of such agreements are not specified by this calling standard.

5.1.7 Establishing Handlers

The list of established frame-based handlers for a thread is defined by the thread's procedure invocation chain. (See Chapter 7 for information on procedure invocations and call chains.)

A procedure descriptor for which `PDSC_FLAGS_HANDLER_VALID` is 1 must specify in `PDSC_RPD_HANDLER_ADDRESS` the procedure value of an exception handler. The exception handler specified by a procedure descriptor is established when the corresponding procedure is added to the invocation chain; that is, when the procedure designated by the descriptor becomes current.

The exception handler remains established as long as that procedure invocation is part of the invocation chain. The handler is revoked when that procedure is removed from the invocation chain; that is, when the procedure invocation designated by the descriptor terminates, either by returning or being unwound.

Thus, the set of frame-based handlers that is established at any moment is defined by the current procedure call chain.

Dynamic activation or deactivation of exception handlers is not defined by this calling standard (and, in fact, is not permitted within the semantics of many language standards). If this capability is required, it must be defined on a language-by-language basis.

Compilers that choose to support this capability can establish language-specific static exception handlers that provide the dynamic exception handling semantics of that language. Such static handlers would be established by means of the procedure descriptor of the establishing procedure. If a language compiler decides to support dynamic activation of exception handlers, it must be prepared to recognize code that intends to use this feature. This requirement results from the need to add appropriate `trapb` instructions and other compile-time considerations necessary to make dynamic exception handling function correctly.

Note

There can be additional protocols and conventions for dynamic exception handling. Such protocols and conventions might be needed, for example, to enable a debugger to perform effectively within the language exception handling environment. Such conventions are driven by the requirements of the languages and the language support utilities, and are not addressed by this calling standard.

5.1.8 Raising Exceptions

This section describes the four ways of raising exceptions.

5.1.8.1 Raising General Exceptions

A thread can raise a general exception in its own context by calling a system library routine defined as follows:

```
exc_raise_exception (ExceptionRecord)
```

Arguments:

`ExceptionRecord`	The address of a primary exception record

Function Value:

None

The `exc_raise_exception()` routine sets `ExceptionAddress` to the address of the invoking call instruction. If `exc_raise_exception()` detects that the exception record passed as the first argument is not a valid exception record, the routine raises the exception `EXC_INVALID_EXCEPTION_RECORD`.

5.1.8.2 Raising General Exceptions Using gentrap

The *Alpha Architecture Reference Manual* defines the `gentrap` PAL call as a mechanism for software to raise hardware-like exceptions at minimum cost. This mechanism is suitable for use in low levels of the operating system or during bootstrapping when only a limited execution environment is generally available.

In a constrained environment, `gentrap` can be handled directly through the SCB vector. In a more complete environment, the `gentrap` parameter is transformed into a corresponding exception code and reported as a normal hardware exception. Low-level software can use this mechanism to report exceptions in a way that is independent of the execution environment. Compiled code can also use this mechanism to raise common generic exceptions more cheaply than would be possible if the code had to make a full procedure call to `exc_raise_exception()`.

The `gentrap` PAL call is defined as follows:

```
gentrap (EXPT_CODE)
```

Argument:

`EXPT_CODE`	Code for the exception to be raised

If the `EXPT_CODE` value is one of the small negative values shown in Table 5–1, that value is mapped to the corresponding POSIX signal, as shown. The signal qualifier is set to `EXPT_CODE`. The signal can be converted to an exception by installing `exc_raise_signal_exception()` as the signal handler. (See Section 5.1.13.) The `gentrap` instruction sets `ExceptionAddress` to the address of the PAL call.

For other values of `EXPT_CODE`, the behavior of `gentrap` is not defined by this standard.

Note that there are no mechanisms to associate any parameters with an exception raised using `gentrap`.

Table 5–1: gentrap EXPT_CODE Values

EXPT_CODE Value	Symbol	Meaning	POSIX Signal
–1	GEN_INTOVF	Integer overflow	SIGFPE
–2	GEN_INTDIV	Integer divide by zero	SIGFPE
–3	GEN_FLTOVF	Floating overflow	SIGFPE
–4	GEN_FLTDIV	Floating divide by zero	SIGFPE
–5	GEN_FLTUND	Floating underflow	SIGFPE
–6	GEN_FLTINV	Floating invalid operation	SIGFPE
–7	GEN_FLTINE	Floating inexact result	SIGFPE
–8	GEN_DECOVF	Decimal overflow	SIGTRAP
–9	GEN_DECDIV	Decimal divide by zero	SIGTRAP
–10	GEN_DECINV	Decimal invalid operand	SIGTRAP
–11	GEN_ROPRAND	Reserved operand	SIGFPE
–12	GEN_ASSERTERR	Assertion error	SIGTRAP
–13	GEN_NULPTRERR	Null pointer error	SIGTRAP
–14	GEN_STKOVF	Stack overflow	SIGTRAP
–15	GEN_STRLENERR	String length error	SIGTRAP
–16	GEN_SUBSTRERR	Substring error	SIGTRAP
–17	GEN_RANGEERR	Range error	SIGTRAP
–18	GEN_SUBRNG	Subscript range error	SIGTRAP
–19	GEN_SUBRNG1	Subscript 1 range error	SIGTRAP
–20	GEN_SUBRNG2	Subscript 2 range error	SIGTRAP
–21	GEN_SUBRNG3	Subscript 3 range error	SIGTRAP
–22	GEN_SUBRNG4	Subscript 4 range error	SIGTRAP
–23	GEN_SUBRNG5	Subscript 5 range error	SIGTRAP

Table 5–1: gentrap EXPT_CODE Values (cont.)

EXPT_CODE Value	Symbol	Meaning	POSIX Signal
–24	`GEN_SUBRNG6`	Subscript 6 range error	`SIGTRAP`
–25	`GEN_SUBRNG7`	Subscript 7 range error	`SIGTRAP`

5.1.8.3 Raising Unwind Exceptions

The mechanism used to raise an unwind exception is described in detail in Section 5.2.

5.1.8.4 Raising Signal Exceptions

Signal exceptions can be raised asynchronously (such as, for notification of a terminal line hangup) or synchronously. The exact circumstances that cause an asynchronous signal exception to be raised vary widely from hardware exception notification to software notification, as in the POSIX-defined `alarm()` routine. Section 5.1.13 contains information on exception and signal handling coexistence.

5.1.9 Search for and Invocation of Exception Handlers

The search for and subsequent invocation of an exception handler begins with the program counter value that indicates the address at which the exception was raised. Generally, a program counter value is associated with a procedure descriptor. (Section 8.1 describes procedure descriptors.) The procedure descriptor provides information needed to identify the procedure containing the code and interpret those parts of the stack frame that are needed to traverse the procedure call chain. (Chapter 7 discusses procedure invocation and call chains.)

If a null frame procedure or other fragment of code does not have an associated procedure descriptor, it is assumed that an appropriate initial program counter value is located in the normal return address register ($26). If there is a procedure descriptor associated with this address, the search for an exception handler begins using that address; otherwise, a fatal exception is raised and the executing thread is terminated.

The next sections discuss the order of invocation for exception handlers as well as handler invocation and arguments.

5.1.9.1 Invocation Order for Exception Handlers

When an exception is raised, established exception handlers are invoked in a specific order.

Any frame-based handlers are invoked in order, from the handler established by the most current procedure invocation to the handler established by the oldest predecessor in the invocation chain.

If no frame-based handlers have been established, or if all of them reraise the exception, the system last chance handler is invoked.

A **nested exception** occurs if an exception is raised while an exception handler is active. When a nested exception occurs, the structure of the procedure invocation chain, from the most recent procedure invocation to the oldest predecessor, contains the following elements in the specified order:

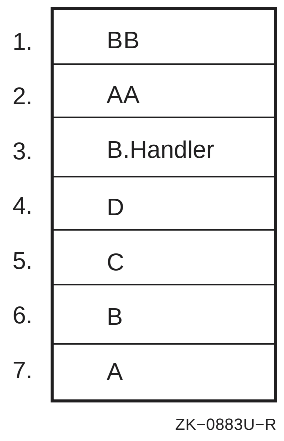
1. The procedure invocation within which the nested exception was raised.
2. Zero or more procedures invoked indirectly or directly by the most recently invoked (most current) handler.
3. The most current handler.

   If there are zero invocations in item 2, this invocation is the same as item 1 (the invocation in which the nested exception was raised). In this case, items 1 and 3 count as a single invocation.
4. The procedure invocation within which the active exception that immediately preceded the nested exception was raised, that is, the invocation in which the exception was raised for which the most current handler was invoked.
5. Zero or more procedure invocations. All handlers established by these procedures have been invoked for the exception that immediately preceded the nested exception and all have reraised that exception.
6. The establisher of the most current handler.

   If there are zero invocations in item 5, this invocation is the same as item 4 (the invocation in which the exception that immediately preceded the nested exception was raised).
7. Zero or more procedure invocations for which no established handlers have yet been invoked.

Figure 5–3 shows an example of a procedure invocation chain, where BB is the most recent invocation and A is the oldest predecessor. The numbers on the left side of the figure correspond to the elements in the previous list.

Figure 5–3: Procedure Invocation Chain



The following list contains the time-ordered events that gave rise to this invocation chain:

- A calls B calls C calls D.
- An exception is raised in D.
- The handlers established by D and by C are invoked in turn and they reraise the exception.
- B.Handler, the handler established by B, is invoked.
- B.Handler calls AA calls BB.
- A nested exception is raised in BB.

Established handlers are invoked in reverse order to the order in which their establishers were invoked. That is, the search of stack frames for procedure invocations that have established handlers is in the order 1 to 7.

If further nested exceptions occur, this procedure invocation chain structure is repeated for those further nested exceptions. Frame-based handlers are invoked according to the order previously listed; that is, from those established by the most current procedure to those established by the oldest predecessor.

The following pseudocode shows the steps for locating and invoking exception handlers. Note that these steps cover only the search of stack frames for a handler proper and do not address the mapping of a POSIX signal to an exception.

1. Let `current_invocation` be the procedure invocation in which the exception was raised.

2. `[loop]`: If `current_invocation` does not establish a handler, go to step `[check-begin]`.
3. Invoke the handler established by `current_invocation`.
4. If the handler returns `ExceptionContinueExecution` or initiates an unwind, exit from these steps.
5. `[check-begin]`: If `current_invocation` is the beginning of the procedure invocation chain, go to step `[last-chance]`.
6. If `current_invocation` is an active handler, let `current_invocation` be the invocation in which the exception was raised that invoked this active handler, and go to step `[loop]`.
7. Let `current_invocation` be the procedure invocation that invoked `current_invocation`.
8. Go to step `[loop]`.
9. `[last-chance]`: Invoke the system last chance handler.

If, during the search for and invocation of frame-based handlers, the exception dispatcher detects that the thread's main stack is corrupt, the following actions occur:

- The `EXCEPTION_STACK_INVALID` flag is set to `1`.
- The search for handlers immediately proceeds to the system last chance handler.

5.1.9.2 Handler Invocation and Arguments

Every exception handler is invoked as a function that returns a status value. The function call is defined as follows:

```
(*ExceptionHandler)
    ( ExceptionRecord, EstablisherFrame,
                          ContextRecord, DispatcherContext )
```

Arguments:

`ExceptionRecord`	The address of a primary exception record.
`EstablisherFrame`	The virtual frame pointer of the establisher (discussed in Section 7.1).
`ContextRecord`	The address of a `sigcontext` structure containing the saved original context at the point where the exception occurred. During an unwind, this argument contains the address of the `sigcontext` structure for the establisher. (The `sigcontext` structure is defined in the file `/usr/include/signal.h`.)
`DispatcherContext`	The address of a control record for the exception dispatcher.

Function Value:

`STATUS`	A value indicating the action to be taken upon handler return. The valid values are `ExceptionContinueExecution` and `ExceptionContinueSearch`. Note: the exception dispatcher allows additional return values from its own exception handlers.

The control record pointed to by `DispatcherContext` provides communication between the handler and the exception dispatcher (the system routine that actually invokes the handler). This record provides information about the establisher. The following list discusses three fields in the record:

- `ControlPC`

  This field contains the program counter (PC) where control left the establisher of the exception handler; that is, the PC of the call instruction or the instruction that caused the exception. This field can be updated by a handler. If a nested exception occurs during unwinding while the handler is still active, the value of the PC used for the establisher will be the updated value of `ControlPC`. This mechanism can be employed to retire nested exception handling scopes that are local to a procedure in order to assure that each is executed only once at most (even in the presence of a nested exception within such a handler). The `ControlPC` value must, however, always be an address within the procedure whose handler is executing.

- `collide_info`

  This field is a quadword whose meaning is determined by the handler. Typically, a language-specific handler will write new values into this field as it processes nested scopes. If a colliding unwind occurs, the dispatcher sets `collide_info` to the value it had for the establisher. A handler knows to read this field when `EXCEPTION_COLLIDED_UNWIND` is 1. This field can be used in the same manner as the `ControlPC` field to retire nested exception scopes, but it does not have to be an address within the current procedure.

- `FunctionEntry`

  This field contains a pointer to the procedure descriptor for the establisher. The field is read-only to exception handlers, except for handlers for the exception dispatcher itself.

5.1.10 Modification of Exception Records and Context by Handlers

The exception records, exception qualifiers, invocation context blocks, and control records that are passed to an exception handler are always allocated in writable memory. Handlers can write to any location in these data structures. The exception records and exception qualifiers that are passed to a handler are copies of the original ones. Modifications to them are seen by other subsequently called handlers (within the limits described later) but do not affect the original data structures.

The effects of a handler-modifying passed exception information are as follows:

- If the `EXCEPTION_NONCONTINUABLE` flag in the primary exception record is changed from 0 to 1, the exception handler that made the modification must not return `ExceptionContinueExecution`, nor can any handler subsequently invoked for the exception return `ExceptionContinueExecution`.

  If `ExceptionContinueExecution` is returned after the `EXCEPTION_NONCONTINUABLE` flag has been changed from 0 to 1, a nested exception is raised with `ExceptionCode` equalling `EXC_STATUS_NONCONTINUABLE_EXCEPTION`, indicating that an attempt was made to continue from a noncontinuable exception. This second exception is also noncontinuable.
- If any flags in `ExceptionFlags` in the primary exception record are modified except as described in the previous item in this list, there is no effect after the exception handler completes its operation. All handlers subsequently invoked for the exception receive a primary exception record with the flags unmodified.

  However, an exception handler must not change the `EXCEPTION_NONCONTINUABLE` flag from 1 to 0.
- If the contents of the record specified by `ContextRecord` or `DispatcherContext` are modified by a handler, except for `ControlPC` and `collide_info`, the results are *unpredictable* and such a handler does not conform to this calling standard.
- Except as specified in the previous items in this list, all changes made to the exception information will be visible to handlers subsequently invoked for the exception. Any other effects of modifying the exception information are not defined by this calling standard.

5.1.11 Handler Completion and Return Value

When an exception handler has finished all its processing, it performs one of the following actions:

- Reraises the exception
- Continues execution of the thread
- Initiates procedure invocation unwinding

Section 5.2 contains a complete description of the unwinding process. This section discusses the other methods of handler completion.

5.1.11.1 Completion by Reraising the Exception

If an exception handler determines that additional handlers should be invoked for the exception because it could not completely handle the exception, the handler can reraise the exception by returning `ExceptionContinueSearch`.

Reraising causes the next exception handler to be invoked. (See Section 5.1.9.1 for details.)

If all exception handlers established by the thread reraise the exception, the system last chance handler is invoked, with system-dependent results.

5.1.11.2 Completion by Continuing Thread Execution

By returning `ExceptionContinueExecution`, an exception handler can continue execution of the thread at the address specified by the continuation PC in the `ContextRecord`, with the context of the interrupted procedure restored.

If `ExceptionContinueExecution` is returned and the `EXCEPTION_NON-CONTINUABLE` flag is 1, a nested exception is raised with `ExceptionCode` equalling `EXC_STATUS_NONCONTINUABLE_EXCEPTION`. This second exception is also noncontinuable.

5.1.11.3 Completion During Unwinding

When an unwind is in progress, the status returned by handlers must be `ExceptionContinueSearch`. Otherwise, `EXC_STATUS_INVALID_DIS-POSITION` is raised; that is, handlers cannot continue during an unwind operation.

5.1.11.4 Completion from Signal Exceptions

The permissibility and effects of continuing from a signal exception are governed by the underlying signal, as specified by the implementation of the POSIX environment.

5.1.12 Other Considerations in Handling Exceptions

This section details certain aspects of the Alpha architecture that have significant implications for exception handling. These aspects are:

- Exception synchronization
- Continuation from exceptions

The rules presented are designed to assure correct operation across all implementations of that architecture. As with all aspects of this calling standard, optimization information may assure correct behavior as if these rules were followed, without appearing to explicitly do so.

Alternative approaches that exploit implementation-specific characteristics are also possible, but are outside the scope of this calling standard.

5.1.12.1 Exception Synchronization

The Alpha hardware architecture allows instructions to be completed in a different order than that in which they were issued. The architecture also allows for exceptions caused by an instruction to be raised after subsequently issued instructions have been completed. Thus, when a hardware exception occurs, the state of the machine cannot be assumed with precision unless it has been guaranteed by bounding the exception range with the appropriate insertion of `trapb` instructions.

The rules for bounding the exception range are as follows:

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

  Rationale: This rule is required because a standard procedure is not allowed to handle traps that it might not have caused.

- If a procedure has an exception handler that does not simply reraise all arithmetic traps caused by code contained directly within that procedure or by any procedure that might have been called while that procedure was *current*, it must issue a `trapb` instruction in the procedure epilogue while it is still the *current procedure*.

  Rationale: This rule is required because handlers established by previous invocations in the call chain might not be able to handle exceptions from a procedure invocation that is no longer active.

- 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. In addition, the handler must be able to recognize exception PC values that represent `trapb` instructions immediately after a call and adjust the depth appropriately.

These rules ensure that exceptions are detected in the context within which exception handlers have been set up to handle them.

However, 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 that is not sensitive to invocation depth, an exception detected while that called procedure is current might have been caused by an instruction issued while the caller was the current procedure. Therefore, 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.

5.1.12.2 Continuation from Exceptions

The Alpha architecture does not guarantee that instructions are completed in the same order in which they were fetched from memory or that instruction execution is strictly sequential. Continuation after some exceptions is possible, but there are restrictions.

Software-raised general exceptions are, by definition, synchronous with the instruction stream and can have a well-defined continuation point. Thus, a handler has the option of requesting continuation from a software raised exception. However, because 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 in a way that is transparent to subsequent code.

Hardware faults on Alpha systems follow rules that, loosely paraphrased, state the following: if the offending exception is fixed, reexecution of the instruction (as determined from the supplied PC) will yield correct results. This generality does not imply that no instructions following the faulting instruction have been executed. (See the *Alpha Architecture Reference Manual* for details.) Hardware faults can therefore be viewed as similar to software-raised exceptions and can have well-defined continuation points.

Arithmetic traps cannot be restarted unless all the information required for a restart is available. The most straightforward and reliable way for software to 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 due to the negative side effects it has on application performance. A more sophisticated technique that requires typically one `trapb` for each basic block is described in the *Alpha Architecture Reference Manual* in the section on imprecise software completion trap modes.

5.1.13 Exception and Signal Handling Coexistence

The procedure-based exception handling facility defined by this calling standard coexists with a global POSIX-style signal facility. The following list describes the features and limitations of such a coexistence.

- The system provides a special signal handler that serves as an interface between a signal facility and a frame-based exception handling facility. The handler is defined as follows:

  ```
  exc_raise_signal_exception
                          (... system-defined-arguments ...)
  ```

  This special signal handler gathers the software and/or hardware information associated with a signal into an exception record where `ExceptionCode.osf_facility` is `EXC_SIGNAL` (ffe0003(hex)), `ExceptionCode.code` is the signal value, and `ExceptionInformation[0]` contains any signal qualifier. This record is passed to the exception-handling support code in a form that can be processed by normal exception handlers. The exception-handling code then proceeds with its normal search and invocation of procedures as described in Section 5.1.2.

  If any handler returns with `ExceptionContinueExecution` (as described in Section 5.1.2), thread execution resumes at the point where it was interrupted by the signal.

  If no exception handlers are located or if all handlers reraise the exception, the system's last chance handler is invoked with *unpredictable* (in this case, system-dependent) results.

- When an application that utilizes procedure-based exception handling is initialized in an environment where signals are also supported, the language subsystem or user should install the special signal handler `exc_raise_signal_exception()` for those signals that normally result directly from the executing code stream. Those signals include `SIGFPE`, `SIGSEGV`, `SIGBUS`, `SIGILL`, `SIGEMT`, `SIGABRT`, `SIGSYS`, and `SIGTRAP`.

  If it is useful, applications and language run-time support code can install `exc_raise_signal_exception()` for other signals, such as `SIGINT`. However, such actions might not be appropriate for signals that result from external autonomous events or signals, like `SIGIO`, that are used for flow control.

- When `exc_raise_signal_exception()` is invoked as a signal handler, the delivered signal may be blocked, along with perhaps others that were specified when it was installed as the handler. Subsequent occurrences of blocked signals are held pending until they are unblocked. Unblocking is a natural result of a procedure-based exception handler returning `ExceptionContinueExecution`, or invoking an unwind operation or `siglongjmp()`. Signals can be explicitly unblocked during handler execution if the handler invokes `sigsetmask()` or `sigprocmask()`.

  Explicit unblocking is necessary if a handler itself might cause a blocked signal to occur. Similarly, explicit unblocking might be needed if the same signal is raised in two threads and must be processed by procedure-based exception handling at the same time; for example, when one thread cannot wait for the other thread to complete its exception handling.

- Application developers should exercise care if their programs directly utilize signal facility support code in any way not otherwise described by this section for any signal having `exc_raise_signal_exception()` installed as its handler. For such applications, the signal facility mechanisms will continue to operate correctly but there might be *undefined* effects on frame-based exception handling of those signals. Some routines to avoid include: `sigaction()`, `sigblock()`, `signal()`, `sigpause()`, `sigsetmask()`, `sigprocmask()`, and `sigvec()`.
- The exception handling capabilities defined in this calling standard are compatible with the use of a separate signal stack. The selection, change, or other use of a separate signal stack is outside the scope of this calling standard.
- Environment restrictions regarding which functions are signal reentrant and which can safely be called from signal catching functions also apply to exception handlers.

5.2 Unwinding

The unwinding capabilities specified in this section support the following:

- Correct and predictable nonlocal `GOTO`s in a multilanguage environment
- Construction of modular, maintainable multilanguage applications

This section discusses the following topics:

- Overview of unwinding (Section 5.2.1)
- Types of unwind operations (Section 5.2.2)
- Types of unwind invocation (Section 5.2.3)
- Unwind initiation (Section 5.2.4)
- Unwinding from inserted code ranges (Section 5.2.5)
- Multiply active unwind operations (Section 5.2.6)
- Unwind completion (Section 5.2.7)
- Unwind coexistence with `setjmp()` and `longjmp()` (Section 5.2.8)

5.2.1 Overview of Unwinding

The term **unwinding**, or **unwind operation**, refers to the action of returning from a procedure or a chain of procedures by some mechanism other than the normal return path. Performing an unwind operation in a thread causes a transfer of control from the location at which the unwind operation is initiated to a **target location** in a **target invocation**. This transfer of control 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 which are being terminated, as well as the handler for the target invocation.

This behavior gives each procedure invocation the chance to perform clean-up processing before its context is lost. These handlers are invoked with an indication that an unwind is in progress. The exception record passed to the target invocation's handler also has `EXCEPTION_TARGET_UNWIND` set to 1.

Once 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.

The results of attempting an unwind operation to any invocation previous to the top-level procedure of a thread are not defined by this calling standard.

Unwinding does not require an exception handler to be active. Unwind operations can be used by languages to implement nonlocal GOTOs.

5.2.2 Types of Unwind Operations

There are two types of unwind requests: general and exit. The following sections describe each type.

5.2.2.1 General Unwind

A **general unwind** transfers control to a specified location in a specified procedure invocation. The target procedure invocation is specified by a frame pointer. (See Section 7.1 for information on procedure invocation.) The target location is specified with an absolute PC value.

When a general unwind is completed, the registers are updated from the invocation context for the target frame. Register $0 obtains its value from the `ReturnValue` argument to the unwind operation. This action allows a status to be returned to the target of the unwind.

5.2.2.2 Exit Unwind

It is valuable for a thread that is terminating execution to be able to clean up its use of shared resources. In a single-threaded process, there might be global resources, such as files, locks, or shared memory, that are shared among processes. For multithreaded processes, global resources as well as process-wide resources like a heap might need to be restored to a known state.

Because of this need to clean up shared resources, exiting by a user-mode thread can be accomplished only by unwinding. A special type of unwind, called an **exit unwind**, performs the following actions to terminate execution:

- Invokes all established frame-based handlers, passing them an exception record that specifies that an exit unwind is in progress.
- Terminates all procedure invocations up to the beginning of the call chain.
- Terminates the execution of the thread.

Threads that use any mechanism for termination other than the normal return process are not considered to be standard and their behavior is undefined.

5.2.3 Types of Unwind Invocations

There are two types of unwind invocations: those initiated while an exception is active and those initiated while no exception is active. This section describes each type.

5.2.3.1 Unwind Operations with No Active Exception

An unwind that is initiated when no exception is active is usually done to perform a nonlocal `GOTO`; that is, to transfer control directly to some code location that is not part of the currently executing procedure or is not statically known. Even this type of unwind operation must provide a mechanism to allow clean-up operations (including restoring a consistent set of register values) of terminated invocations to be performed. The unwind mechanism supports such clean-up operations.

5.2.3.2 Unwind Operations During an Active Exception

By initiating an unwind operation, the handler, or any descendant procedure called directly or indirectly by the handler, can continue execution of the thread at a location different from the one where the exception was raised.

An unwind operation specifies a target invocation in the procedure invocation chain and a location in that procedure. The operation terminates all invocations up to the target invocation and continues thread execution at the specified location in that procedure.

Before control is transferred to the target location, the unwind operation invokes each frame-based handler that was established by any procedure invocations being terminated, and also invokes the handler for the target invocation.

5.2.4 Unwind Initiation

A thread can initiate a general unwind operation by calling one of two system library routines. The routines differ only in the way their first argument specifies the target frame: as a **virtual frame pointer** or a **real frame pointer**. (Section 7.1 discusses ways to refer to procedure invocations.) These routines are defined as follows:

```
exc_unwind (VirtualTargetFrame, TargetPC,
                            ExceptionRecord, ReturnValue)

exc_unwind_rfp (RealTargetFrame, TargetPC,
                            ExceptionRecord, ReturnValue)
```

Arguments:

`VirtualTargetFrame`	If nonzero, specifies the virtual frame pointer of the target procedure invocation to where the unwind is to be done. If zero, specifies that an exit unwind is initiated and causes the `EXCEPTION_EXIT_UNWIND` flag to be set to 1 in the exception record.
`RealTargetFrame`	If nonzero, specifies the real frame pointer of the target procedure invocation to which the unwind is to be done. If zero, specifies that an exit unwind is initiated and causes the `EXCEPTION_EXIT_UNWIND` flag to be set to 1 in the exception record.
`TargetPC`	Specifies the address within the target invocation at which to continue execution. If a target frame argument is zero, this argument is ignored.
`ExceptionRecord`	If nonzero, specifies the address of a primary exception record. If zero, specifies that a default exception record should be supplied.
`ReturnValue`	Specifies the value to use as the return value (contents of $0) at the completion of the unwind.

Function Value:

None

If the `ExceptionRecord` argument is zero, `exc_unwind()` or `exc_unwind_rfp()` supplies a default exception record. That default exception record specifies exactly one exception record in which `ExceptionCode` is `EXC_STATUS_UNWIND`. For an explicit or default exception record, the `EXCEPTION_UNWINDING` flag is set to 1; and, if a null target frame argument is specified, the `EXCEPTION_EXIT_UNWIND` flag is set to 1. The `ExceptionAddress` is set to `TargetPC`.

If the `ExceptionRecord` argument is specified when the unwind is initiated, all other properties of the exception record are determined by `ExceptionRecord`. If `exc_unwind()` or `exc_unwind_rfp()` detects that a specified exception record is not a valid unwind record, the routine will raise the exception of `EXC_INVALID_EXCEPTION_RECORD`. If the frame corresponding to the target frame argument cannot be found, the system last-chance handler is called because all procedures have been terminated.

Once an unwind is initiated, control never returns from the call.

5.2.5 Unwinding from an Inserted Code Range

An **inserted code range** is any number of contiguous instructions that are added to a procedure. These instructions were not part of the original procedure as it was written. The compiler can create inserted code ranges by inlining, and a post-link tool can create inserted code ranges through object modification. One or more code range descriptors (CRD) are used to describe these inserted code ranges.

The inserted code ranges are also described by a run-time procedure descriptor (RPD). These code ranges can have stack, local variables, and exception handlers associated with them. Additional information is provided in the RPD of the inserted code range in the form of a `return_address` offset to support unwinding the frame from within these code ranges.

Unwinding from code whose RPD has a nonzero `return_address` field is the same as unwinding from code whose RPD has a zero `return_address` field, with one exception: after all other actions are performed, the new current PC is set to the address calculated from the `return_address` field in the RPD.

See Section 8.1 for more information on code range descriptors and run-time procedure descriptors.

5.2.6 Multiply Active Unwind Operations

Sometimes, an unwind operation is initiated while another unwind is already active. Such a situation could occur if a handler that is invoked during 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, in turn, initiates another unwind operation.

An unwind that is initiated while a previous unwind is active is either a nested unwind or a colliding unwind. This section discusses both types of multiply active unwind operations.

5.2.6.1 Nested Unwind

A **nested unwind** operation is an unwind that is initiated while a previous unwind is active. For a nested unwind, the target invocation in the procedure invocation chain is not a predecessor of the most current active unwind handler. That is, 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. 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.

5.2.6.2 Colliding Unwind

Like the nested unwind, a **colliding unwind** is an unwind that is initiated while a previous unwind is active. For a colliding unwind, the target invocation in the procedure invocation chain is a predecessor of the most current active unwind handler. That is, a colliding unwind terminates one or more procedure invocations that would have been terminated by the previously active unwind.

A colliding unwind is detected when the most current active unwind handler is terminated. This detection of a colliding unwind is referred to as a **collision**. When a collision occurs, the second (more recent) unwind operation takes precedence and the previous unwind is abandoned.

The next action is to reinvoke the most current established handler because its establisher has not been unwound. The `EXCEPTION_COLLIDED_UNWIND` flag is set in the exception record to indicate this situation to the handler.

5.2.7 Unwind Completion

When an unwind is completed, the following conditions are in effect:

- 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 that existed when that invocation was last current, except for the contents of scratch registers.
- The GP register contains a pointer to the GOT that is appropriate to the target procedure.
- The $0 register contains the return value that was passed by the routine which invoked the unwind.
- Execution continues at the target location.

5.2.8 Unwinding Coexistence with setjmp and longjmp

The procedure invocation unwinding facility defined by this calling standard can coexist and interoperate with the `setjmp()` and `longjmp()` facilities. It is sufficient for the `jmp_buf` array to consist of the frame pointer and program counter values that are needed as arguments to `exc_unwind()` or `exc_unwind_rfp()`. A null pointer can be provided for the `ExceptionRecord` argument and the value of the `longjmp()` expression can be provided for the `ReturnValue` argument.

Any environment that conforms to this calling standard must implement nonlocal `GOTO`s by using `exc_unwind()` or `exc_unwind_rfp()` (or an equivalent means) to allow all procedures being terminated to clean up any local or global states, as appropriate.

Note

The `longjmp()` routine for Tru64 UNIX does not use an unwind operation. Therefore, in the presence of frame-based exception handling, it is preferable to use `exc_longjmp()`, implemented through an unwind operation.

6 Stack Limits in Multithreaded Execution Environments

This chapter discusses the following topics:

- Stack limit checking (Section 6.1)
- Stack overflow handling (Section 6.2)

The focus of these discussions is on dealing with stack limits in a multithreaded environment; however, the same information applies to singlethreaded environments. Although this calling standard is compatible with a multithreaded execution environment, the detailed mechanisms, data structures, and procedures that support this capability are not specified in the standard.

For a multithreaded 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 the code that implements the multithreaded mechanism itself.
- Exception handling is fully reentrant and multithreaded.
- There are three ways to terminate a thread correctly:
  - By returning from the initial procedure in which the thread began execution
  - By a call to `exc_unwind()` or `exc_unwind_rfp()`, specifying a null target environment
  - By using some other method that includes correct thread termination involving unwind processing for all of the active frames of that thread

6.1 Stack Limit Checking

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

Detection of a stack overflow condition is important. If a stack overflow is not detected, a thread that is writing into what it considered to be stack storage could modify data allocated in that memory for some other purpose. The results of such a situation would most likely be unpredictable and undesirable. In some cases, the overflow could result in unreproducible application failures.

Checking for stack overflow is a requirement for procedures that might execute in a multithreaded environment.

6.1.1 Stack Region Definitions

The various stack regions are defined as follows:

new stack region	This region of the stack extends from the old value of SP–1 to the new value of SP.
stack guard region	In a multithreaded environment, the memory beyond the limit of each thread's stack is protected by contiguous **guard pages**, which form the stack's **guard region**.
stack reserve region	In some cases, it is desirable to maintain a stack **reserve region**. This region is a minimum-sized region that is immediately above a thread's guard region. A reserve region is useful to ensure that the following conditions exist: • Exceptions or asynchronous signals have stack space to execute on a thread's stack • The exception dispatcher and any exception handler it might call have stack space to execute after an invalid attempt to extend the stack has been detected. The Tru64 UNIX calling standard does not require a reserve region.

6.1.2 Methods for Stack Limit Checking

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

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

Note

An access can be performed using a load operation or a store operation; however, care must be taken to use an instruction that is guaranteed to make an access to memory. For example, do not use an `ldq $31,*` instruction because the Alpha architecture allows it to result in no memory access at all, rather than a memory read access whose result is discarded because of the $31 destination.

There are two methods for stack-limit checking: implicit and explicit. In addition, the stack reserve region can be checked. The following sections describe each type of checking.

6.1.2.1 Implicit Stack Limit Checking

There 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, 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, the stack can be extended by an increment that is equal in size to one-half the guard region without any further accesses.

Generally, the stack frame layout (shown in Section 3.1.2) and entry code rules (described in Section 3.2.6.1) do not make it feasible to guarantee access to the lowest address of a new stack region without introducing an extra access solely for that purpose. Consequently, this calling standard uses the second strategy. Although the maximum amount of implicit stack extension is smaller, the check is achieved at no additional cost.

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

These factors are the basis for the following rule: If the stack is being extended by an amount less than or equal to 4096 and no reserve region is required, no explicit stack-limit checking is required.

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

- Explicit stack limit checking must be performed unless the amount by which SP is decremented is known to be less than or equal to 4096 and no reserve region is required.
- Some byte in the new stack region must be accessed before SP can be decremented for a subsequent stack extension. This access can be performed before or after SP is decremented for this stack extension, but it must be done before SP can be decremented again.
- No standard procedure call can be made before some byte in the new stack region is accessed.
- 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 SP is decremented, but before the access in the new stack region occurs.

These conventions ensure that the stack pointer will not be decremented so far that it points to accessible storage beyond the stack limit without having the error detected by one of the following:

- The guard region being accessed by the thread
- An explicit stack limit check failure occuring

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. Note that the size of the guard region and the number of guard pages is defined by the system, not by this calling standard.

6.1.2.2 Explicit Stack Limit Checking

If the stack is being extended by an amount that is unknown at compile time or of a known size greater than the maximum implicit check size (4096), 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 smaller 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 SP4096 because, in the absence of more specific information, the previous guaranteed access relative to the current stack pointer might 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.

A simple algorithm that satisfies these rules (but can result in twice the minimum number of accesses) calls for performing a sequence of accesses in a loop starting with the previous value of SP and then decrementing by the minimum no-check extension size (4096) up 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 which is unknown at compile time or of a known size greater than the minimum implicit check size (4096) must test new stack segments (as described previously) in a loop that does not modify SP. The procedure prologue must then update the stack with one instruction that copies the new stack pointer value into 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 specially tailored to be called from a procedure prologue.

6.1.3 Stack Reserve Region Checking

The size of the stack reserve region, if one exists, must be included in the increment size used for stack limit checks. However, the size is not included in the amount by which the stack is actually extended. Depending on the reserve size, stack reserve region checking could completely eliminate the ability to use implicit stack limit checking.

6.2 Stack Overflow Handling

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

- The system transparently extends the thread's stack, resets the TEB stack limit value appropriately, and continues execution of the thread.
- The signal `SIGSEGV` is generated. If `exc_raise_signal_exception()` is installed as the handler for `SIGSEGV`, the corresponding exception is raised. (See Section 5.1.13 for information on exception and signal-handling coexistence.) To provide enough stack space to execute

the exception dispatcher and handlers, specify that the `sigstack` be used when `SIGSEGV` is delivered. (See the `sigstack`(2) reference page.)

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 as well as by the handling of exceptions.

7 Procedure Invocations and Call Chains

This chapter discusses the library routines that support procedure call tracing. These routines are used to:

- Refer to a given procedure invocation (Section 7.1)
- Provide the context of a procedure invocation (Section 7.2)
- Navigate (walk) the procedure call chain (Section 7.3)

The chapter also describes the data structures and procedures that these routines require.

7.1 Referencing a Procedure Invocation

When a reference to a specific procedure invocation is made at run time, the virtual frame pointer or the real frame pointer for that invocation can be used. The **virtual frame pointer** of a procedure invocation is the contents of the stack pointer at the entry point of the procedure. The **real frame pointer** of a procedure is the contents of the stack pointer after the size of the fixed part of the stack frame has been subtracted from the virtual frame pointer.

Note that the virtual frame pointer of an invocation is not the value used by the procedure itself for addressing. The contents of the SP register are modified in the procedure prologue and the resulting real frame pointer value is then sometimes copied into FP (as in the case of a variable size stack frame). The real frame pointer is always used for addressing local storage throughout the remainder of the procedure.

The real frame pointer is not, by itself, sufficient to unambiguously identify all possible procedure invocations. For example, a null frame procedure has the same real frame pointer as its caller because the null frame procedure allocates no stack storage. This ambiguity is of no consequence for the purposes of this calling standard because the real frame pointer value is always used in combination with a program counter value that identifies an instruction within a particular procedure.

The static link used in calling nested procedures in languages such as Pascal and Ada is usually the virtual frame pointer or the real frame pointer value. The choice is implementation-dependent and can vary from language to language and release to release.

The full context of a specific procedure invocation is provided through the use of the `sigcontext` data structure. The `sigcontext` structure is defined in the file `/usr/include/signal.h`.

7.2 Providing a Procedure Invocation Context

A thread can obtain its own context by calling a system library function defined as follows:

```
exc_capture_context (ContextRecord)
```

Arguments:

`ContextRecord`	Address of a `sigcontext` structure into which the procedure context of the caller is written

A thread can obtain the invocation context of the procedure preceding another procedure context by calling a system library routine defined as follows:

```
exc_virtual_unwind (FunctionEntry, ContextRecord)
```

Arguments:

`FunctionEntry`	Address of the function table entry for the function. If zero, the function table entry is looked up using the PC from `ContextRecord`.
`ContextRecord`	Address of a `sigcontext` structure. The given structure is updated to represent the context of the previous (calling) frame.

Function Value:

`InPrologueOrEpilogue`	If 1, indicates that the resulting program counter value in the given `ContextRecord` is within the prologue or the epilogue code of the function. If zero, indicates that the program counter is in the body of the function.

The `exc_virtual_unwind()` procedure takes a `sigcontext` structure together with its associated procedure descriptor and updates the context to reflect the state of the caller at the point where it made the call.

7.3 Walking the Call Chain

During program execution, it is sometimes necessary to navigate (walk) the call chain. For example, frame-based exception handling requires call chain navigation. Call-chain navigation is possible only in the reverse direction; for example, latest-to-earliest procedure, or top to bottom procedure.

There are two steps for performing call chain navigation:

1. Build a `sigcontext` structure when given a program state that contains a register set.

   For the current routine, an initial `sigcontext` structure can be obtained by calling `exc_capture_context()`.

2. Repeatedly call `exc_virtual_unwind()` until the end of the chain is reached.

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

No assumptions should be made 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.

8 Procedure Descriptors

Procedure descriptors serve the following functions:

- They provide the means for mapping from an arbitrary program counter value to the descriptive information associated with the code at that address.
- They provide information about a procedure, such as which registers are saved, where they are saved, and the length of the prologue. This information is needed for call-chain navigation in general and exception handling in particular.
- They provide a means for link-time, post-link-time, and execution-time tools to identify all of the instructions that constitute a complete procedure.
- They provide a means for link-time, post-link-time, and execution-time tools to distinguish code from data.

Every procedure, except for null frame procedures, must have an associated procedure descriptor. (Null frame procedures are discussed in Section 3.1.4.)

Note

The term **procedure descriptor** also appears in the manual *Symbol Table/Object File Specification*. The use of the term in that manual refers to a structure defined in the file `/usr/include/sym.h` and should not be confused with procedure descriptors used for exceptions, as described in this manual.

This chapter covers the following procedure-descriptor topics:

- Representation (Section 8.1)
- Access routines (Section 8.2)
- Use with run-time generated code (Section 8.3)

8.1 Procedure Descriptor Representation

Procedure descriptors on Tru64 UNIX for Alpha systems use two kinds of structures: code range descriptors and run-time procedure descriptors.

Code range descriptors associate a contiguous sequence of addresses with a run-time procedure descriptor. This mapping can be many-to-one.

Run-time procedure descriptors provide descriptive information about a procedure or part of a procedure.

The creation of procedure descriptors involves the combined actions of compilers and assemblers, the linker, and the loader. The sections that follow focus solely on the result of this composite process and describe the resulting descriptors as seen by the executing run-time environment. The figures present the physical representation of the procedure descriptor structures. Macros, which have the prefix `PDSC_`, can be used to access the fields of these structures. The physical representation and the macros are defined in the file `/usr/include/pdsc.h`.

A *single* run-time procedure descriptor should be used to describe a complete procedure, even if it has multiple entry points, as in FORTRAN. In this case, each entry point typically consists of the following elements:

- Prologue
- Entry-point specific code
- Branch to a common join point following the last entry point

Although the prologues might contain different code, they must all achieve the same effect: the same registers saved at the same offset, the same frame size, and so on.

Two instructions are said to be *part of the same procedure* if they are described by the same run-time procedure descriptor (RPD). However, if code has been inserted into a procedure (for example, through inlining or instrumentation), this code may have a separate RPD associated with it. In this situation, the `return_address` field of the procedure descriptor will have a nonzero value. Before determining whether an instruction within an inserted code range is in the same procedure as another instruction, the RPD chain must be followed until an RPD with a `return_address` field equal to zero is found. Then, the comparison can be done with the top-level RPD. See Section 5.2.5 for more information.

Notes on binary compatibility:

1. Earlier versions of this Calling Standard used a separate procedure descriptor for each entry point of a procedure. This method is now obsolete. However, tools must be able to handle applications compiled under the obsolete model as well as the current model.
2. Exception handling will operate correctly on both obsolete and current models of RPDs.

3. The property that two instructions are part of the same procedure only if they have the same procedure descriptor cannot be used on applications compiled under the obsolete model.
4. Using a single RPD to describe a complete procedure was implemented in Tru64 UNIX Version 5.1. Earlier versions used the obsolete model.

8.1.1 Code Range Descriptors

The code-range table is an array of code-range descriptors, as shown in Figure 8–1.

The following macros are used to access data in code range descriptors:

`PDSC_CRD_BEGIN_ADDRESS`

Specifies the address of the beginning of a code range. The elements of the array are sorted so that the portion of the address space covered by a single element starts at the `PDSC_CRD_BEGIN_ADDRESS` value contained in that element and extends to, but does not include, the `PDSC_CRD_BEGIN_ADDRESS` value encoded in the next successive element. Thus, the last array element provides the end address of the last code range and does not start a new range.

`PDSC_CRD_PRPD`

Specifies the address of the associated run-time procedure descriptor. A `PDSC_CRD_PRPD` value of zero indicates a null frame procedure for which an implicit run-time procedure descriptor is assumed with the characteristics described in Section 3.1.4.

`PDSC_CRD_CONTAINS_PROLOG`

Indicates whether the code range begins with a procedure prologue.

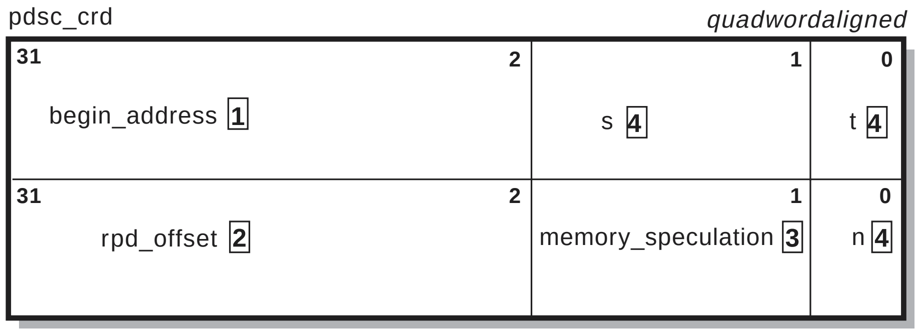
`PDSC_CRD_MEMORY_SPECULATION`

Indicates that memory traps (`SIGSEGV`, `SIGBUS`) raised in this procedure should not be delivered.

`PDSC_CRD_TYPE_STANDARD`
`PDSC_CRD_TYPE_CONTEXT`
`PDSC_CRD_TYPE_NON_CONTEXT`
`PDSC_CRD_TYPE_NON_CONTEXT_STACK`
`PDSC_CRD_TYPE_DATA`

Indicate the procedure context of the code described by the code range descriptor.

Figure 8–1: Code Range Descriptor



ZK-0866U-AI

1 `begin_address`

Contains a longword that is the offset from the base of the code range table to the starting point of the code to which the associated run-time procedure descriptor applies. The low two bits of this longword are reserved for use as flags; they must be masked out before the containing longword is used as an offset.

2 `rpd_offset`

Contains a longword that is the self-relative offset to the associated run-time procedure descriptor. The low two bits of this longword are used as flags; they must be masked out before the containing longword is used as an offset. An `rpd_offset` value of zero indicates a null frame procedure for which an implicit run-time procedure descriptor is assumed with the characteristics defined in Section 3.1.4.

3 `memory_speculation` (bit 1 of `rpd_offset`)

Indicates that memory traps (`SIGSEGV`, `SIGBUS`) occurring in this procedure should not be delivered.

4 `s`, `t`, `n` (bits 1,0 of `begin_address`, bit 0 of `rpd_offset`)

These three bits are used together to identify the context of the code that the code range descriptor is describing, listed below. If the `rpd_offset` is 0, then the values of `s`, `t`, and `n` are zero as well.

In the following descriptions for the types of code range descriptor, the occurrence of `CONTEXT` in the name implies that the associated procedure is current for exception handling purposes during that part of the procedure specified by the CRD; similarly, `NON_CONTEXT` implies the procedure is not current. `STACK` implies that stack storage has been allocated. A very common special case in indicated by the occurence of `STANDARD` in the name because it corresponds to a standard sequence of up to three states. `DATA` implies storage that is not executable code at all.

- `CRD_TYPE_STANDARD (s=0, t=0, n=0)`

  This is the standard range descriptor. It consists of the prologue up to, and including, the SP-setting instruction (non_context), the prologue after the SP-setting instruction (non_context_stack), and the primary (context) code range. The fields `sp_set` and `entry_length` are relative to this segment. There can only be one `CRD_TYPE_STANDARD` code range for each procedure.
- `CRD_TYPE_CONTEXT (s=0, t=0, n=1)`

  This is a full routine context descriptor, but it does not contain the prologue.
- `CRD_TYPE_DATA (s=0, t=1, n=0)`

  This code range describes data that resides within the text area.
- `CRD_TYPE_NON_CONTEXT (s=0, t=1, n=1)`

  This code range is not in a routine context, and it does not contain stack allocation that needs to be deallocated.
- `CRD_TYPE_NON_CONTEXT_STACK (s=1, t=0, n=1)`

  This is a non-context region for exception handling purposes, but the stack has been allocated.

The encodings (s=1, t=0, n=0), (s=1, t=1, n=0) and (s=1, t=1, n=1) are reserved. One of these encodings should be used for the future extension of the CRD.

Note on binary compatibility:

1. In earlier versions of this Calling Standard, bits 0 and 1 of `begin_address` were undefined, and bit 0 of `rpd_offset` was set if no prologue was associated with the code range. The encodings of `s`, `t`, and `n` maintain the binary compatibility of the original specification. Be aware that the macro `PDSC_CRD_CONTAINS_PROLOG` will return TRUE if bit 0 of `rpd_offset` is clear.

8.1.2 Run-Time Procedure Descriptors

Run-time procedure descriptors provide information about a procedure needed for exception handling and other tools. Table 3–1 lists this information. There are two forms: long and short. Both forms encode the same information. The short form is used to save space for the most commonly occurring cases. Inserted code must use the long form of the RPD. Figure 8–2 shows the long form; Figure 8–3 shows the short form.

Each figure shows two alternative representations for the first longword. The first representation applies to stack frame procedures and is shown in the main part of the figure. The second representation applies to register frame procedures and is shown as a separate longword at the end of the figure. The `PDSC_FLAGS_REGISTER_FRAME` flag, which is one of the flags common to both forms, determines which representation applies.

Descriptions of the physical fields follow the figures. These descriptions include the calculations used to obtain the logical field from the physical field. Most fields are common to both procedure descriptor forms. The long form has three additional fields and two additional flags as shown in Figure 8–2. The additional fields are `entry_ra`, `return_address`, and a field reserved for future use. The additional flags are `PSDC_FLAGS_ARITHMETIC_SPECULATION` and `PDSC_FLAGS_EXTENDER`.

Figure 8–2: Long Form Run-Time Procedure Descriptor

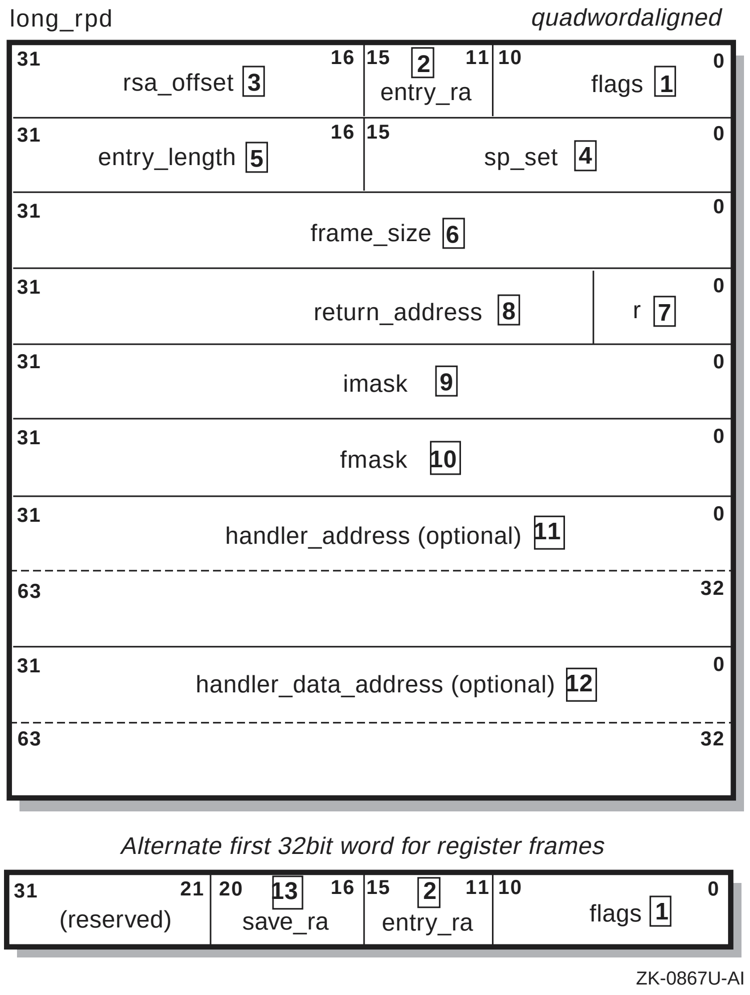
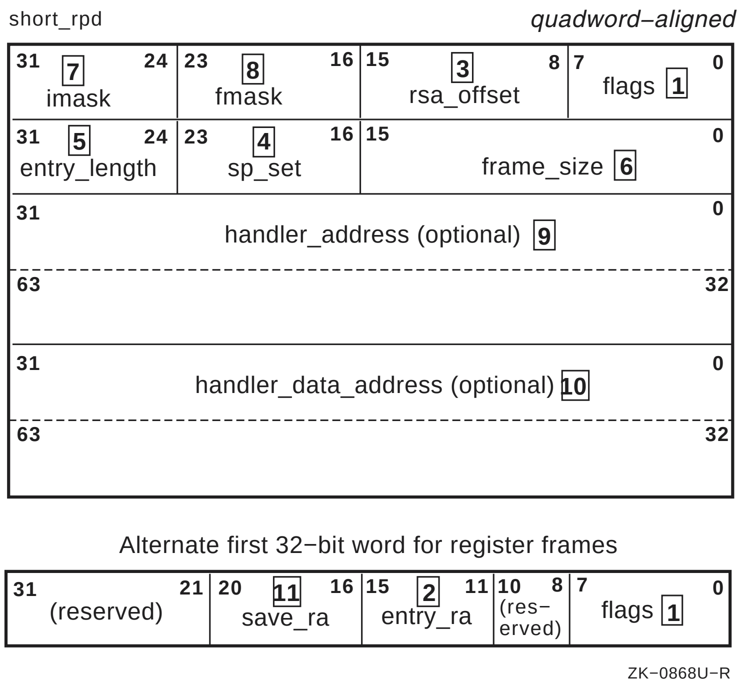


Figure 8–3: Short Form Run-Time Procedure Descriptor



$\boxed{1}$ The `flags` field represents `PDSC_RPD_FLAGS` and encodes the following flags:

- `PDSC_FLAGS_SHORT` (bit 0) is 1 if and only if this structure is a short form run-time procedure descriptor.
- `PDSC_FLAGS_REGISTER_FRAME` (bit 1) is 1 only for a register frame procedure.
- `PDSC_FLAGS_BASE_REG_IS_FP` (bit 2) is 1 if and only if $15 is used as the frame pointer.
- `PDSC_FLAGS_HANDLER_VALID` (bit 3) is 1 if and only if an exception handler's address and handler data are specified using the `PDSC_RPD_HANDLER` and `PDSC_RPD_HANDLER_DATA` fields.
- `PDSC_FLAGS_EXCEPTION_MODE` (bits 4, 5, and 7) encodes the caller's desired exception reporting behavior when calling certain mathematically oriented library routines. The 3-bit integer value is formed with bit 7 as the MSB and bit 4 as the LSB. The possible values for this field and the corresponding meanings are as follows:
  - `PDSC_EXC_SILENT`(0)

    Raise no exceptions and create only finite values (no infinities, denormals, or NaNs). In this mode, the function result or the

C language errno must be examined for any error indication. This mode is the default.

  - PDSC_EXC_SIGNAL(1)

    Raise exceptions for all error conditions except for underflow, which yields a zero result.

  - PDSC_EXC_SIGNAL_ALL(2)

    Raise exceptions for all error conditions (including underflows).

  - PDSC_EXC_IEEE(3)

    Raise no exceptions (except as controlled by separate IEEE exception enable bits), and create infinite, denormal, and NaN values according to the IEEE floating-point standard.

  - PDSC_EXC_CALLER(4)

    Perform the exception mode behavior specified by this procedure's caller.

- PDSC_FLAGS_EXCEPTION_FRAME (bit 6) is 1 for a frame that includes a hardware exception context.
- PDSC_FLAGS_ARITHMETIC_SPECULATION (bit 8) is 1 if arithmetic traps (SIGFPE) ocurring in this procedure should not be delivered.
- PDSC_FLAGS_EXTENDER (bit 10) is reserved for future indication of an extended form of run-time procedure descriptor.

2 The entry_ra field is the number of the register in which the return address is passed to this procedure. The entry_ra field can be accessed with PDSC_RPD_ENTRY_RA.

3 The rsa_offset field is the signed difference in quadwords between the stack frame base (SP or FP as indicated by PDSC_FLAGS_BASE_REG_IS_FP) and the register save area. (See Section 3.1.2 for information on stack frame procedures.)

```
PDSC_RPD_RSA_OFFSET = rsa_offset * 8
```

4 The sp_set field is the unsigned offset in instructions (longwords) from the entry address of the procedure to the single instruction in the procedure prologue that modifies the stack pointer. This offset must be zero when there is no such instruction because the procedure has a PDSC_RPD_FRAME_SIZE of 0.

```
PDSC_RPD_SP_SET = sp_set * 4
```

5 The entry_length field is the unsigned offset in instructions (longwords) from the entry address to the first instruction in the procedure code segment following the procedure prologue.

```
PDSC_RPD_ENTRY_LENGTH = entry_length * 4
```

$\boxed{6}$ The `frame_size` field is the unsigned size in quadwords of the fixed portion of the stack frame for this procedure.

```
PDSC_RPD_FRAME_SIZE = frame_size * 8
```

The value of SP at entry to this procedure can be calculated by adding `PDSC_RPD_FRAME_SIZE` to the value SP or FP, as indicated by `PDSC_FLAGS_BASE_REG_IS_FP`. `PDSC_RPD_FRAME_SIZE` cannot be 0 for a stack frame procedure because the stack frame must include space for the register save area.

Note: If a procedure needs a frame size that is too large to be represented using the `frame_size` field, a variable-size stack frame should be used. In this case, the FP register is used to address a fixed size area that needs to be just large enough to include the preserved state. An arbitrarily large stack area can then be covered by the SP register.

$\boxed{7}$ The lower two bits of the `return_address`, *`r`*, are reserved for future use. These lower two bits must be masked off the containing longword before the offset is used.

$\boxed{8}$ The `return_address` contains a longword that is the offset from the base of the code range table to the point in the code where the flow of control should return when the code range described by this RPD is complete. The low two bits of this longword are reserved and must be masked out before the containing longword is used as an offset. If this field is not equal to 0, then the code segment that this RPD describes is an inserted code range. See Section 5.2.5 for more information.

```
PDSC_RPD_RETURN_ADDRESS_FIELD = return_address & (~3)

PDSC_RPD_RETURN_ADDRESS =
     code_range_table_base_address + PDSC_RPD_RETURN_ADDRESS_FIELD
```

$\boxed{9}$ The `imask` field is a bit vector (0 – 31) specifying the integer registers that are saved in the variable portion of the register save area on entry to the procedure. The least significant bit corresponds to register $0. Bits 31, 30, 28, and the register containing the entry return address of this mask should never be set because $31 is the integer Read-As-Zero register, $30 is the hardware SP, $29 (GP) is always assumed to be destroyed during a procedure call or return, and the return address is saved at known offset zero in the register save area in every stack frame procedure. The `imask` field can be accessed with `PDSC_RPD_IMASK`.

$\boxed{10}$ The `fmask` field is a bit vector (0 – 31) specifying the floating-point registers that are saved in the variable portion of the register save area on entry to the procedure. The least significant bit corresponds to register $f0. Bit 31 of this mask should never be set because it corresponds to the floating-point Read-As-Zero register. The `fmask` field can be accessed with `PDSC_RPD_FMASK`.

11 The `handler_address` field is an absolute procedure value (quadword) for a run-time static exception handling procedure. The `handler_address` field can be accessed with `PDSC_RPD_HANDLER`.

This part of the procedure descriptor is optional. However, it must be supplied if `PDSC_FLAGS_HANDLER_VALID` is 1. If `PDSC_FLAGS_HANDLER_VALID` is 0, the contents or existence of `PDSC_RPD_HANDLER` is *unpredictable*.

12 The `handler_data_address` field is a quadword of data for the exception handler. The `handler_data_address` field can be accessed with `PDSC_RPD_HANDLER_DATA`.

This part of the procedure descriptor is optional. However, it must be supplied if `PDSC_FLAGS_HANDLER_VALID` is 1. If `PDSC_FLAGS_HANDLER_VALID` is 0, the contents or existence of `PDSC_RPD_HANDLER_DATA` is *unpredictable*.

13 The `save_ra` field is the number of the register in which the return address is maintained during the body of the procedure. The `save_ra`field can be accessed with `PDSC_PRD_SAVE_RA`.

If this procedure uses the standard calling conventions and does not modify \$26, both `PDSC_RPD_ENTRY_RA` and `PDSC_RPD_SAVE_RA` will specify \$26.

The short form run-time procedure descriptor differs from the long form in the following ways:

- An `entry_ra` of \$26 is assumed for the return address register in a stack frame procedure (`PDSC_FLAGS_REGISTER_FRAME` is 0).
- The `rsa_offset` field is limited to at most 255 quadwords (2040 bytes).
- The `fmask` field represents only registers \$f2 through \$f9.
- The `imask` field represents only registers \$8 through \$15. (Note that \$8 is not normally a preserved register.)
- The `frame_size` field is limited to at most 65,535 quadwords (524,280 bytes).
- The `sp_set` offset is limited to at most 255 instructions (1020 bytes).
- The `entry_length` offset is limited to at most 255 instructions (1020 bytes).
- A `return_address` value of 0 is assumed because the `return_address` field does not exist in the short form. This includes the low-order reserved bits.

If any of these restrictions cannot be satisfied, the long form run-time procedure descriptor must be used.

8.1.3 Examples

This section contains two examples of how to use code range descriptors and procedure descriptors. The first example involves multiple entry points and the second example involves instrumented code.

8.1.3.1 Multiple Entry Point Example

Consider the following routine, written in a hypothetical C-like language that allows multiple entry points.

```
double sd;
double ent1(int i, double d) {

LX: if (i || ((int)d) < 5) {
        return foo() * 2.0 + PI;
    }
     return 0.0;

/* Alternate entry point */
double ent2(int i, double d):
    double d2;
    if (!i) goto LX;

    if ( (d2 = foo2()) != 0.0) {
        return tailcallee(d2*d2+sd);
    }
    return sd * 10;
}
```

A sample version of code is annotated below. This case is very contrived to show many types of descriptors.

```
                                   CRD#       Types               Notes
    .globl  ent1
    .ent    ent1
ent1:
    ldgp    $gp, 0($27)               0      non-context           (1)
    bne     $16, lab1                 0
    lda     $sp, -32($sp)             0

    cvttq   $f17, $f16                1      non-context-stack
    stt     $f16, ($sp)               1
    ldq     $0, ($sp)                 1
    cmplt   $0, 5, $1                 1
    bne     $1, lab2                  1
    lda     $sp, 32($sp)              1

lab0:
    fclr    $f0                       2      non-context           (2)
    ret     $26                       2
lab1:
    lda     $sp, -32($sp)             3      standard              (5)
lab2:
    stq     $26, ($sp)                3                            (6)

    .prologue 1
    .frame  $sp,32,$26
    ...call to foo...                 3
    ldq     $f1, 2.0                  3
    ldq     $26, ($sp)                3
    lda     $sp, 32($sp)             3

    mult    $f0, $f1, $f0            4      non-context           (2)
    addt    $f0, $f16, $f0           4
    ret     $26                      4

    .globl  ent2
    .aent   ent2

ent2:
    bne     $16, lab0                5      non-context           (1)

lab3:
    ldgp    $gp, 0($27)              5                            (4)
    lda     $sp, -32($sp)            5

    stq     $26, ($sp)               6      non-context-stack

    .prologue 1
    .frame  $sp,32,$26
    ...call to foo2...               7      context
    ldq     $f16, sd                 7
    fbeq    $f0, lab5                7
    mult    $f0, $f0, $f0            7
    ldq     $26, ($sp)               7
    lda     $sp, 32($sp)             7

    addt    $f0, $f16, $f16          8      non_context           (2)
    br      tailcallee               8                            (3)

lab5:
    ldq     $f0, 10.0                9      context
    lda     $sp, 32($sp)             9

    mult    $f0, $f16, $f0          10      non_context           (2)
    ret     $26
```

Notes:

1. The code from `ent1` through to `lab1` is a **shrinkwrap**, where some code is moved outside of full procedure context, as is the `bne` instruction following `ent2`. In this very contrived example, the savings are minimal for the first case (only the return register save is avoided).
2. This code is a floating exit sequence where the stack deallocation was scheduled away from the corresponding return. See Section 3.2.6.2.1 for more information.
3. This is a tailcall, where control does not transfer back to the caller.
4. In this hypothetical case, the compiler chose to prefer the `bne` over being able to skip the gp prologue for the entry.
5. If the routine has a stack frame, it is an error for a primary prologue to not contain a stack adjustment because the adjustment is required in order to get a correct `sp_set` value.

The following tables show the code range descriptors and the run-time procedure descriptors associated with the preceding example.

Code range descriptors:

crd	begin_address	crd_type	rpd_offset
CRD0	ent1	non_context	PD0
CRD1	ent1+12	non_context_stack	PD0
CRD2	lab0	non_context	PD0
CRD3	lab1	standard	PD0
CRD4	lab2+20	non_context	PD0
CRD5	ent2	non_context	PD0
CRD6	lab3+8	non_context_stack	PD0
CRD7	lab3+12	context	PD0
CRD8	lab3+36	non_context	PD0
CRD9	lab5	context	PD0
CRD10	lab5+8	non_context	PD0

Run-time procedure descriptors:

rpd	sp_set	entry_length	frame_size	return_address
PD0	0	2	4	0

8.1.3.2 Instrumented Code Example

The following example displays an instrumentation point inserted by ATOM, a post-link tool. The standard “Hello world” test case is instrumented with the ATOM tool Third Degree.

```
main()
{
        printf("Hello world!\n");
}
```

The following program listing is a disassembly of `main`. The code range inserted by a post-link tool begins at 0x120063984 and ends at 0x1200639ac, inclusively.

```
                                                 CRD#   Type             Notes
      main:
0x120063978: ldah    gp, 8186(r27)                 0    standard          (1)
0x12006397c: lda     gp, 18184(gp)                 0
0x120063980: lda     sp, -16(sp)                   0
0x120063984: lda     sp, -48(sp)                   1    non-context       (2)
0x120063988: stq     r16, 24(sp)                   2    non-context-stack
0x12006398c: stq     r17, 16(sp)                   2
0x120063990: stq     r26, 8(sp)                    2
0x120063994: lda     r16, 4(r31)                   2
0x120063998: lda     r17, 48(sp)                   2
0x12006399c: bsr     r26, 0x1200069f0(r31)         2
```

```
0x1200639a0: ldq     r16, 24(sp)                 2
0x1200639a4: ldq     r17, 16(sp)                 2
0x1200639a8: ldq     r26, 8(sp)                  2
0x1200639ac: lda     sp, 48(sp)                  2
0x1200639b0: stq     r26, 0(sp)                  3    non-context-stack (3)
0x1200639b4: ldq     r16, -32736(gp)             4    context
0x1200639b8: ldq     r27, -32608(gp)             4
0x1200639bc: jsr     r26, (r27), 0x1200625d4(r31)  4
0x1200639c0: ldah    gp, 8186(r26)               4
0x1200639c4: lda     gp, 18112(gp)               4
0x1200639c8: bis     r31, r31, r0                4
0x1200639cc: ldq     r26, 0(sp)                  4
0x1200639d0: lda     sp, 16(sp)                  4
0x1200639d4: ret     r31, (r26), 1               4    implicitly non-   (4)
                                                        context
```

Notes:

1. This CRD is still of type `CRD_TYPE_STANDARD` because `sp_set` is still relative to this code range. The `entry_length` field is no longer relative to this code range, and should be zero.
2. This instruction has a type of `CRD_TYPE_NON_CONTEXT` because the previous stack allocation was for a different RPD.
3. This instruction has a type of `CRD_TYPE_NON_CONTEXT_STACK` because the stack was allocated at `main+8` and this is still part of the prologue.
4. The return instructions at the end of the procedure is implicitly non-context because of the reserved instruction sequence rules defined in Section 3.2.6.2.1. Using an additional CRD here would be redundant, but not incorrect.

The following tables show the code range descriptors and the run-time procedure descriptors associated with the preceding example.

Code range descriptors:

crd	begin_address	crd_type	rpd_offset
CRD0	main	standard	PD0
CRD1	main+12	standard	PD1
CRD2	main+16	non_context_stack	PD1
CRD3	main+56	non_context_stack	PD0
CRD4	main+58	context	PD0

Run-time procedure descriptors:

rpd	sp_set	entry_length	frame_size	return_address
PD0	2	0	2	0
PD1	0	0	6	CRD3

For the purpose of comparison, this is the non-instrumented version of `main`:

```
[hw.c:  1] 0x120001120:    27bb2000     ldah    gp, 8192(r27)
[hw.c:  1] 0x120001124:    23bd6f60     lda     gp, 28512(gp)
[hw.c:  1] 0x120001128:    23defff0     lda     sp, -16(sp)
[hw.c:  1] 0x12000112c:    b75e0000     stq     r26, 0(sp)
[hw.c:  3] 0x120001130:    a61d8020     ldq     r16, -32736(gp)
[hw.c:  3] 0x120001134:    a77d80a0     ldq     r27, -32608(gp)
[hw.c:  3] 0x120001138:    6b5b7b05     jsr     r26, (r27), 0x11ffffd50(r31)
[hw.c:  3] 0x12000113c:    27ba2000     ldah    gp, 8192(r26)
[hw.c:  3] 0x120001140:    23bd6f44     lda     gp, 28484(gp)
[hw.c:  4] 0x120001144:    47ff0400     bis     r31, r31, r0
[hw.c:  4] 0x120001148:    a75e0000     ldq     r26, 0(sp)
[hw.c:  4] 0x12000114c:    23de0010     lda     sp, 16(sp)
[hw.c:  4] 0x120001150:    6bfa8001     ret     r31, (r26), 1
```

The following tables show the code range descriptors and run-time procedure descriptors for the preceding example.

Code range descriptors (noninstrumented):

crd	begin_address	crd_type	rpd_offset
CRD0	main	standard	PD0

Run-time procedure descriptors (noninstrumented):

rpd	sp_set	entry_length	frame_size	return_address
PD0	2	4	2	0

8.2 Procedure Descriptor Access Routines

A thread can obtain information from the descriptor of any procedure in the thread’s virtual address space by calling system library functions.

In the course of running and debugging a program, there are times when it is necessary to identify which procedure is currently executing. During normal thread execution, the current procedure must be determinable any time an exception arises so that the proper handlers will be invoked. In addition, a debugger must know which procedure invocation is currently executing so it can obtain information about the current state of the execution environment.

To determine precisely the current execution context, two pieces of information are required:

- The procedure that is currently executing

- Which instance of that procedure is currently executing

This context of the current procedure and the specific instance of that procedure invocation are referred to as the current procedure invocation or simply, current procedure. At any point in the execution of a thread, only one procedure is considered to be the current procedure.

In this calling standard, the value in the PC is used to indicate the current procedure by means of the code range table described in Section 8.1.1.

The following system-supplied routine is used to obtain the address of the procedure descriptor that corresponds with any given PC value within the current address space.

```
exc_lookup_function_entry (ControlPC)
```

Arguments:

`ControlPC`	Specifies a PC value in the current address space for which the procedure value is to be returned.

Function Value:

`PROC_DESC`	Specifies the address of the code range descriptor for the procedure containing the requested PC. If the return value is null, the PC is not currently mapped.

The following system-supplied routine is used to obtain the address of the base of the code range array for the procedure descriptor that corresponds B to any given PC value within the current address space.

```
exc_lookup_function_table (ControlPC)
```

Arguments:

`ControlPC`	Specifies a PC value in the current address space for which the code range base address is to be returned.

Function Value:

`PROC_CRD`	Specifies the address of the base of the code range descriptor array for the procedure descriptor of the procedure containing the requested PC. If the return value is null, the PC is not currently mapped.

At times, it is useful to acquire the GOT segment value for a procedure; that is, the value of the GP register. The following system-supplied routine is used to obtain the GP value corresponding to any given PC value within the current address space.

```
exc_lookup_gp (ControlPC)
```

Arguments:

`ControlPC`	Specifies a PC value in the current address space for which the GP value is to be returned.

Function Value:

`GP_VALUE`	Specifies a GP value for the procedure containing the requested PC. If the return value is null, the PC is not currently mapped.

8.3 Run-Time Generated Code

Code generated at run time is important for applications that include:

- Interactive languages
- Software bit block transfers (for efficient support of graphic displays that do not provide hardware bit block transfers)
- String pattern matching
- Sorting
- Interpretive execution and instruction stream modification by programming and debugging tools
- Construction of bound procedure variables that have a representation consistent with that of simple procedure values

To maintain stack traceability when code generated at run time is executed, procedure descriptors must be provided for that code. Such procedure descriptors must describe correctly the characteristics of the code and the environment within which that code executes.

Before run-time generated code that uses any exception facilities (directly or indirectly) can be executed, system library functions must be called to communicate the code ranges, procedure descriptors, and GP values to the execution environment. This communication is accomplished by calling the following two system-supplied routines:

```
exc_add_pc_range_table (PROC_DESC_ADDR, LENGTH)
```

Arguments:

`PROC_DESC_ADDR`	Specifies the base address of the code range array for the procedure descriptors.
`LENGTH`	Specifies the number of code range elements in the array.

An exception is raised if the `exc_add_pc_range()` operation cannot be completed successfully.

```
exc_add_gp_range (BEGIN_ADDRESS, LENGTH, GP_VALUE)
```

Arguments:

`BEGIN_ADDRESS`	Specifies the first address for which `GP_VALUE` applies.
`LENGTH`	Specifies the number of bytes from `BEGIN_ADDRESS` for which `GP_VALUE` applies.
`GP_VALUE`	Specifies the GP value to be associated with the addresses in the range `BEGIN_ADDRESS .. BEGIN_ADDRESS + LENGTH + 1`.

An exception is raised if the `exc_add_gp_range()` operation cannot be completed successfully.

When procedure information is no longer valid or if the code will not be executed again, two system library routines should be called to remove the procedure mapping information. These system library routines are defined as follows:

```
exc_remove_pc_range_table (PROC_DESC_ADDR)
```

Arguments:

`PROC_DESC_ADDR`	Specifies the base address of the code range array for the procedure descriptors.

An exception is raised if the `exc_remove_pc_range_table()` operation cannot be completed successfully.

```
exc_remove_gp_range (BEGIN_ADDRESS)
```

Arguments:

`BEGIN_ADDRESS`	Specifies the beginning address for which GP-value information should be removed.

An exception is raised if the `exc_remove_gp_range()` operation cannot be completed successfully.

The following steps show how run-time code should be constructed and released:

1. Allocate memory for the code.
2. Write the code and any procedure descriptors to memory.
3. Call `exc_add_pc_range_table()` and `exc_add_gp_range()`.
4. Invoke an `imb` (instruction memory barrier) operation as required by the Alpha architecture.
5. Execute the code.
6. Call `exc_remove_pc_range_table()` and `exc_remove_gp_range()`.

7. Deallocate the memory containing the code.

Index

A

access routine
  procedure descriptor, 8–16
active state, 1–5
address
  calling computed, 3–15
  data alignment, 4–12
  definition of, 1–4
address representation, 2–1
address value, 2–1
alarm routine, 5–14
aligned record layout, 4–13, 4–14
alignment
  data, 4–12
  faults, 4–12
Alpha architecture, 2–4
Alpha hardware, 1–2
Alpha system, 1–1
  (*See also* calling standard)
Alpha systems, 1–1
  (*See also* Alpha architecture)
  hardware exceptions, 5–4
argument home area
  definition of, 4–4
  description of, 3–6
argument item
  definition of, 3–13
  description of, 4–1
  in an argument list, 4–11
  locations of, 4–2
  passing arguments with, 4–1, 4–2
  representation of, 4–2
  unused bits, 4–5
argument item sequence
  contents of, 4–5
  definition of, 3–13
  description of, 4–2
argument list
  call convention for, 3–13
  definition of, 1–4
  description of, 4–2
  homed memory structure, 4–3
  normal structure, 4–2
  with function value return by
    reference, 4–11
  with high-level languages, 4–5
argument passing
  mechanisms for, 4–1, 4–8
argument register
  description of, 2–2

B

base register
  architecture, 2–4
  description of, 2–4
bit array, 4–14
bit field, 4–14
bit string, 4–14
bits
  unused in passed data, 4–5
bound procedure
  definition of, 1–4
bound procedure value
  definition of, 2–1
  discussion of, 3–16
byte
  definition of, 1–6

C

call
conventions, 3–12
call chain
interpreted by the procedure descriptor, 3–1, 5–15
navigation of, 7–2
call frame
as maintained by procedures, 3–1
definition of, 1–4
code
run-time, 8–18
code range descriptor
definition of, 8–2
description of, 8–2
code range table, 8–3
code sequence
entry, 3–17
exit, 3–17
colliding unwind operation, 5–29
compiler, 1–1
computed call, 3–15
continuation
from exception, 5–22
control
flow of, 3–1
transfer of, 3–11
conventional saved register
description of, 2–2
conventional scratch register
description of, 2–2
current procedure, 3–19, 3–23, 5–10
description of, 8–17
entry code sequence, 3–17
exit code sequence, 3–23
identification of, 8–16
with trapb instruction, 5–21

D

data
alignment address, 4–12
alignment of, 4–12
allocation of, 4–11
manipulation of, 4–1
passing, 4–1
passing unused bits, 4–5
returning, 4–9
sending, 4–7
descriptor, 1–5
(*See also* procedure descriptor)
argument passing with, 4–2
code range, 8–2
definition of, 1–5
function value return mechanism, 4–11
procedure, 3–1, 8–1
sending data, 4–9
detail flag, 5–7
double-precision complex value, 4–3

E

entry code
register frame procedure example, 3–20
stack frame procedure example, 3–19
entry code sequence
discussion of, 3–17
environment flag, 5–7
event processing, 5–1
exc_add_gp_range routine, 8–18, 8–19
exc_add_pc_range_table routine, 8–18, 8–19
exc_capture_context routine, 7–2, 7–3
exc_longjmp routine, 5–30
exc_lookup_function_entry routine, 8–17
exc_lookup_function_table routine, 8–17
exc_lookup_gp routine, 8–17

exc_raise_exception routine, 5–12
exc_raise_signal_exception routine, 5–13, 5–23, 5–24, 6–5
exc_remove_gp_range routine, 8–19
exc_remove_pc_range_table routine, 8–19
exc_unwind routine, 5–27, 5–28, 5–30, 6–1
exc_unwind_rfp routine, 5–27, 5–28, 5–30, 6–1
exc_virtual_unwind routine, 7–2, 7–3
exception
  bounding a range, 5–21
  continuation from, 5–22
  general, description of, 5–4
  general, hardware-raised, 5–22
  general, raising, 5–12
  general, software-raised, 5–22
  hardware, 5–9
  hardware-caused, 5–4
  nested, 5–15
  raising, 5–11
  signal, 5–5
  software-caused, 5–4
  synchronization of, 5–21
  unwind, 5–4
exception address, 5–8
exception code, 5–5, 5–7
exception condition
  definition of, 1–5
exception dispatcher
  communicating with exception handler, 5–18
  description of, 5–2
exception flag, 5–7
exception handler
  arguments for, 5–17
  completion of, 5–3, 5–19
  current procedure for, 3–19
  deactivation of, 5–11
  definition of, 1–5
  dynamic activation of, 5–11
  establishing, 5–10
  frame-based, 5–10, 5–15, 5–16
  invocation of, 5–2, 5–8, 5–14, 5–16, 5–17
  invocation order for, 5–14
  location of, 5–16
  return value for, 5–19
  search for, 5–14
  system last chance, 5–15, 5–20
exception handling
  coexistence with signal handling, 5–22
  discussion of, 5–1
  overview, 5–2
exception information, 5–9
exception qualifiers, 5–3
  number of, 5–9
exception record, 5–8
  discussion of, 5–6
  general exception, 5–9
  modification of, 5–18
  signal exception, 5–9
  unwind exception, 5–9
exception value, 5–3
exit code
  register frame procedure example, 3–24
  reserved instruction sequence for, 3–21
  stack frame procedure example, 3–24
exit code sequence
  discussion of, 3–20
  steps in, 3–23
exit unwind operation, 5–25

F

fixed temporary location, 3–6
flag
  detail, 5–7
  environment, 5–7
  exception, 5–7
floating-point data
  passing of, 4–3
floating-point register
  argument items mapped to, 4–2
  description of, 2–3
  passing data in, 4–3
  restoring, 3–8
  storing, 3–8
  storing contents in homed area argument structure, 4–4
flow control, 3–1
frame base register
  with register save area, 3–7
frame pointer
  conventions, 3–19
  definition of, 3–19
  real, 5–27, 7–1
  value of, 3–19
  virtual, 5–27, 7–1
frame-based exception handler
  description of, 5–10
  establishing, 5–10
  invoking, 5–16
function
  definition of, 1–5
function result
  call convention for, 3–13
function value register
  description of, 2–2
function value return
  by descriptor, 4–11
  by immediate value, 4–10
  by reference, 4–10

G

general exception
  (*See* exception)
gentrap instruction
  definition, 5–12
  discussion of, 5–12
  EXPT_CODE values for, 5–13
global offset table
  computing the address of a segment, 3–14
  description of, 2–4
  with procedures, 3–15
global pointer, 2–4
global pointer register
  description of, 2–2
GOT
  (*See* global offset table)
GP
  (*See* global pointer)
granularity
  of memory, 4–13
guard pages, 6–2
guard region
  definition of, 6–2
  minimum size of, 6–3
  relative to stack extension, 6–3

H

handler
  (*See* exception handler)
hardware exception
  associated exception information, 5–9
  definition of, 1–5
  description of, 5–4
  relationship to gentrap, 5–12
  rules for, 5–22
hardware register
  (*See* floating-point register, integer register, register)
hardware state
  exceptional, 1–5
heavyweight procedure, 3–4
homed argument list structure, 4–3

I

image
 definition of, 1–5
 program layout, 2–4
immediate value
 argument passing with, 4–1
 definition of, 1–5
 function value return mechanism, 4–10
 sending data, 4–8
integer register
 argument items mapped to, 4–2
 description of, 2–2
 passing arguments in, 4–8
 passing of argument items in, 4–3
 storing, 3–8
 storing contents in homed area argument structure, 4–4
invocation
 exception handler, 5–14, 5–17
invocation chain
 (*See* call chain)
invocation procedure, 7–1

J

jacket procedure
 construction of, 3–16
 purpose of, 3–12
jmp_buf array, 5–30

L

language support procedure
 definition of, 1–6
layout
 conventions for records, 4–13
library procedure
 definition of, 1–6
lightweight procedure, 3–10
link-time optimization, 3–15
linkage, 3–14
longjmp routine, 5–30
 similarity to exception handler invocation, 5–8
 with unwind operations, 5–30
longword
 definition of, 1–6

M

memory
 granularity of, 4–13
mprotect function, 3–17
multithreaded environment
 stack limits in, 6–1

N

native user mode, 1–2
natural alignment
 definition of, 1–6
nested exception
 description of, 5–15
 raised by exception handler, 5–3
nested unwind operation, 5–29
new stack region, 6–2
nonparametric argument, 4–8
nonscratch register, 3–10
null argument, 4–5
null frame procedure
 definition of, 3–1
 discussion of, 3–11
 with exception handler search, 5–14

O

octaword
 definition of, 1–6
optimization
 at link time, 3–15

P

parametric argument, 4–9
performance, 1–1
primary exception, 5–6
procedure
  bound, 1–4
  current, 3–17, 3–19, 3–23, 5–10, 8–16, 8–17
  definition of, 1–6
  descriptors for, 8–1
  heavyweight, 3–4
  jacket, 3–12, 3–16
  language support procedure, 1–6
  library procedure, 1–6
  lightweight, 3–10
  null frame, 3–1, 3–11, 5–14
  properties of, 3–2
  register frame, 3–1, 3–10
  representation of, 2–1
  stack frame, 3–1, 3–4
  standard conforming, 1–7
  stub, 3–12
  types of, 3–1
  without frames, 3–11
procedure call chain
  (*See* call chain)
procedure descriptor, 5–14, 8–1
  access routine, 8–16
  code range, 8–2
  current with frame-based exception handler, 5–10
  definition of, 1–6
  description of, 2–1
  discussion of, 8–1
  overview, 3–1
  providing information for exception handling, 5–11
  representation of, 8–1
  run-time, 8–2, 8–5
  run-time generated code, 8–18
  specifying exception handler for, 5–11
procedure invocation, 7–1
  chain structure, 5–15
  providing context for, 7–2
  referencing, 7–1
procedure prologue
  computing GOT pointer value, 3–14
  contents of, 3–19
  discussion of, 3–18
  execution of, 3–17
  length of, 3–19
procedure value
  bound, 2–1, 3–16
  call convention for, 3–12
  definition of, 1–6
  description of, 2–1
  simple, 2–1
  when calling a procedure, 3–14
procedure value register
  description of, 2–2
process
  definition of, 1–6
program image
  (*See* image)
prologue
  (*See* procedure prologue)

Q

quadword
  definition of, 1–6

R

ReadAsZero register
  description of, 2–2
real frame pointer, 7–1
record, 5–6
  (*See also* exception record)
  layout conventions, 4–13
reference
  argument passing with, 4–1
  definition of, 1–7
  function value return mechanism, 4–10
  sending data, 4–8, 4–9

register, 2–3
(*See also* floating-point register; integer register)
floating-point, 2–3, 3–8
integer, 2–2, 3–8
names, 2–4
nonscratch, 3–10
scratch, 3–10
usage conventions, 2–2
register frame procedure
definition of, 3–1
discussion of, 3–10
entry code example, 3–20
exit code example, 3–24
register save area, 3–7
discussion of, 3–7
layout of, 3–8
registers
naming conventions, 2–4
usage conventions, 2–2
reserve region, 6–2
reserved instruction sequence, 3–21
return address
call convention for, 3–12
return address register
description of, 2–2
return of function value
by descriptor, 4–11
by immediate value, 4–10
by reference, 4–10
run-time code
construction and release of, 8–19
run-time environment, 1–2
run-time generated code, 8–18
run-time procedure descriptor
description of, 8–2
discussion of, 8–5

S

saved register
conventional, 2–2
scratch register, 3–10
conventional, 2–2
volatile, 2–2
secondary exception, 5–6
setjmp routine
with unwind operations, 5–30
sharable image
definition of, 1–7
sigaction routine, 5–24
sigblock routine, 5–24
sigcontext structure, 7–2, 7–3
siglongjmp routine, 5–23
signal
definition of, 1–7
signal exception
description of, 5–5
exception records for, 5–9
raising, 5–14
signal handler, 5–13
signal handling
coexistence with exception handling, 5–22
signal routine, 5–3, 5–24
sigpause routine, 5–24
sigprocmask routine, 5–23, 5–24
sigsetmask routine, 5–23, 5–24
sigvec routine, 5–24
simple procedure
description of, 3–16
simple procedure value
definition of, 2–1
single-precision complex value, 4–3
sink register
description of, 2–2
software exception
continuation of, 5–22
description of, 5–4
stack
access by simple procedure, 3–16
contents of, 3–13

extension of, 6–3, 6–4, 6–5, 6–6
growth of, 3–13
guard region, 6–2
limit, 6–1
limit, checking, 6–1, 6–2
limit, checking, explicit, 6–4
limit, checking, implicit, 6–3
new region, 6–2
overflow of, 6–1, 6–5
passing of argument items in, 4–3
pointer, 6–1
reserve region, 6–2
reserve region, checking, 6–5
resetting, 3–21
usage, 3–13
stack base register, 3–4, 3–5
stack frame, 3–19
contents of, 3–4
description of, 3–4
fixed size, 3–4
variable size, 3–5
stack frame base register
description of, 2–2
stack frame procedure, 3–4
definition of, 3–1
discussion of, 3–4
entry code example, 3–19
exit code example, 3–24
format of, 3–4
stack guard region, 6–2
stack limit, 6–1
(*See also* stack)
checking, 6–1, 6–2
checking, explicit, 6–4
checking, implicit, 6–3
stack pointer, 6–1, 6–5
contents for virtual frame pointer, 7–1
contents of, 3–13
setting for use with unwind support routines, 3–20
used to calculate frame pointer, 3–19
with stack limit checking, 6–4
stack pointer register
description of, 2–2
stack reserve region, 6–2
checking, 6–5
stack return, 4–11
stack temporary area, 3–7
stack usage
call convention for, 3–13
standard call
definition of, 1–7
standard conforming procedure
definition of, 1–7
status value, 5–5
stub procedure
purpose of, 3–12
synchronization
of exception, 5–21

T

target invocation, 5–25
target location, 5–25
TEB
(*See* thread environment block)
thread, 1–7
definition of, 1–7
terminating, 6–1
thread environment block (TEB), 6–1
thread-safe code
definition of, 1–8
threads
multithreaded environment, 6–1
trapb instruction
for arithmetic traps, 5–22
for bounding exception range, 5–21
to establish current procedure, 5–21
with exception handler, 5–21
with exception handling, 5–11

U

undefined
  definition of, 1–8
unpredictable
  definition of, 1–8
unwind exception
  description of, 5–4
  exception records for, 5–9
  raising, 5–14
unwind operation
  colliding, 5–29
  completion of, 5–29
  discussion of, 5–24
  exit, 5–25
  exit code sequences with, 3–22
  for unblocking signals, 5–23
  general, 5–25
  initiating, 5–27
  invoking, 5–26
  multiply active, 5–28
  nested, 5–29
  overview, 5–24
  restoring a stack during, 3–21
  with longjmp routine, 5–30
  with setjmp routine, 5–30
usage hint, 3–21

V

va_list, 4–4
virtual frame pointer, 7–1
volatile scratch register
  description of, 2–2

W

word
  definition of, 1–6