Digital Equipment Corporation
ALPHA Calling Standard

Revision: 2.0

27–April–1990

Issued by:

Chip Nylander and Terry Grieb, Software Development Technologies

digital™

Digital Equipment Corporation

CONTENTS

FIGURES

TABLES

Preface

This document defines the ALPHA calling standard. There are two flavors of this standard; one for all 64-bit address ALPHA environments called the *ALPHA-64 Calling Standard* and one for a 32 bit address ALPHA/VMS environment called the *ALPHA-32 Calling Standard*

```
\\
The standard is NOT under ECO control yet.

This version of the calling standard had two major goals:

 o To provide a stability point for the ALPHA-32 flavor
 o To provide a complete specification for the ALPHA-64 flavor

There is still an outstanding issue for both flavors of the calling standard
concerning what connection, if any, there will be between UNIX style signal
handling and exception handling.

The ALPHA-32 flavor of the calling standard is otherwise complete and any
further technical changes will be discussed with all parties affected. In some
cases clarifications and additional information may be added.

With the exception noted above, the ALPHA-64 flavor of the calling standard
is also complete. It should be noted that while this flavor of the standard
represents a complete solution to a problem, it has not been verified for
ALPHA/ULTRIX. Nor have all the ramifications for ALPHA/VMS phase II been
investigated. Both of these area will be examined in the future as plans
in each of these areas solidify.

We solicit your review and opinions about all aspects of the contents of this
document. Spelling errors and other syntactic comments as well as technical
questions and requests for clarification should be made via the notes file
mentioned below. Please use separate notes for syntactic comments and technical
questions. If you have concerns or issues with either flavor of the calling
standard you should raise them in a timely fashion.

Any strong concerns or objections about approach or content of this document
should be directed to all of the key participants in the ALPHA development
process. We suggest that the notes file is an appropriate media for expression
of such concerns. Significant change will only result from forging a new
consensus or majority view among the key participants.

Additional copies of this document may be obtained by mailing the following
information to TLE::ALPHA_CALLSTD

        Name:
        Mailstop:
        Enet Address:
        Project:
        Organization:

Material will NOT be distributed without COMPLETE information.

A restricted notes conference (TURRIS::ALPHA_CALLSTD) is available for issues
relating to this standard. Membership requests should be made to
TLE::ALPHA_CALLSTD
\\
```

Acknowledgements

It should be noted that besides the authors the following people contributed significant amounts of time and energy towards the creation of this document. Their efforts and technical contributions are appreciated by all.

- Rich Grove
- Ron Brender
- Steve Hobbs
- Peter Craig
- Kim Peterson
- Bill Noyce
- Gary Barton

And of course the various others that have provided "special" review.

Revision History

Date	Revision Number	Author	Summary of Changes
30-Jun-1989	0.1	C. Nylander	Original Draft, with complete Introduction, 32/64 Common, and 32-bit specific sections, and incomplete 64-bit specific section.
14-Jul-1989	0.2	T. Grieb	Changed document to reflect 64-bit argument lists for the ALPHA-32 calling standard and changed some text to clarify certain areas based on first round review comments.
26-Jul-1989	0.3	T. Grieb	Changed stack alignment, Rearranged frame descriptors, more general cleanup, added null descriptor type, changed ALPHA-64 call standard dynamic return, added IS_A_DESC concept.
13-Sep-1989	1.0	T. Grieb	Fixed entry and exit code sequences, Made VIRTUAL_ORIGIN a quadword, Fixed some incorrect offsets in diagrams, Changed register assignments, Changed size of some fields in frame descriptor and 64-bit descriptors, Clarified some wording/definitions having to do with alignment, Added IEEE floating data types, Made T2 and VT2 data types unsupported for 64-bit calling standard, added exception handler context quadword and made exception handling storage optional, Added preliminary thread section, Added SP as base register concept to stack frame descriptor, added argument zero/sign extend information, small doc reorg to localize some discussions, removed FREEZE/THAW and WAIT, added preliminary implementors appendix, Changed DRAIN to DRAINT, Rearranged frame descriptor formats again, changed layout of register save area, Defined DRAINT usage requirements, Release (Halloween Version)
08-Nov-1989	1.1	T. Grieb	Fixed FORTRAN descriptor return value, changed names of CHF$x_ symbols, Merged in 64-bit exception handling stuff, added invocation handle concept, Modified a lot of the multithread text, generalized procedure value usage.
19-Jan-1990	1.2	T. Grieb	MAJOR document rewrite to separate out 64 bit calling standard and make it PC mapped, changed AI register layout, changed frame descriptor to procedure descriptor, changed procedure descriptor layouts, added all exception handling stuff for ALPHA-64, moved some of TEB format to appendix, added data types.
19-Jan-1990	1.3	T. Grieb	General cleanup after limited review. Establish base version for discussions
14-Feb-1990	1.4	T. Grieb	More cleanup after review, Update for Rev 3 ALPHA SRM
16-Mar-1990	1.5	T. Grieb	Bump version after "release". More data passing fine tune. Wording cleanup, Document now reflects some interoperability issues, Update argument passing.
26-Mar-1990	1.6	T. Grieb	Bump version after "release",Final cleanup for "Limited Review" (April Fool's version)

Date	Revision Number	Author	Summary of Changes
04-Apr-1990	1.7	T. Grieb	Bump version after "release", Add signature information, reorg AI register and small shuffle of procedure descriptors, added more flag bits, change name to ALPHA, changed "condition" to "exception", changed other names and various things for MIPS compatibility, added MIPS compatibility appendix, added more text to PC map design appendix, removed H float as supported data type
23-Apr-1990	2.0	T. Grieb	Bump version for next public document (May Day version).

Document Organization

This standard is organized into the following distinct parts:

1. Part I, ALPHA-64 Calling Standard, contains specifications that apply to all programs executing in the 64-bit address ALPHA user mode environment (referred to as *ALPHA-64*) under any operating system that supports this standard and that environment.
2. Part II, ALPHA-32 Calling Standard, contains specifications that apply to all programs executing in the 32-bit address ALPHA user mode environment (referred to as *ALPHA-32*) under the ALPHA/VMS operating system.

   This section is written as *differences* from existing specifications (including the VAX/VMS calling standard and those specifications in Part I). Therefore the appropriate background reading is required for full understanding of the material in Part II.
3. Part III, Appendices, contains topics that are related to calling standard but are not part of the formal specification.

Notation Used in This Document

The specifications in this document are presented as follows:

- Editorial Comment

  ```
  \\
  All text enclosed in double backslashes, illustrated by this paragraph, is
  editorial comment, is not formally a part of the specification, and will not
  necessarily be in future revisions of this document.
  \\
  ```

- Constants

  Constants are presented symbolically with their value given at the point of definition in this standard. Concrete language bindings for each constant are provided in system definition files external to this standard.

  ```
  \\
  Note that the symbols used in this document do NOT follow the current VAX/VMS
  naming conventions nor do they adhere to any other POSIX or other convention.
  We need to decide what we want the ALPHA naming conventions to be (e.g. are
  future naming conventions affected by VIP and the presence of open, standard
  APIs on VMS, ULTRIX and OZIX), and then this document will be brought into
  conformance.
  \\
  ```

- Functional Interfaces

  Functional interface syntax is presented in abstract form. Concrete language bindings for each functional interface are provided in system definition files external to this standard.

  The semantic capabilities of each functional interface are defined in American Language.

- Algorithms

  Algorithms are presented precisely, as a series of steps, in American Language.

- Conventions

  All conventions that are important to correct program execution are presented in a form appropriate to each convention.

- Methods

  Actual or recommended methods are presented informally, using examples, suggestions, or other appropriate form.

- Numbering

  All numbers are represented as decimal values unless otherwise indicated. Non decimal numbers are typically represented with the name of the base in parentheses following then number, E.G. 1B(Hex).

- Figures

  Figures that represent memory or register layouts follow the convention that increasing addresses run from the top to bottom and right to left of a page. Most significant bits are on the left and least significant bits are on the right.

- Code Examples

All code examples in this document are supplied strictly for purposes of explanation. They are presented in a form that expresses the relevant concept with clarity. They do *not* reflect optimized and properly scheduled code sequences that a compiler would generate.

Assembler syntax used follows the *ALPHA System Reference Manual* (including Appendix A, section A.4.3 on Stylized Code Forms) and does not represent actual ALPHA Assembler notation which has yet to be defined.

- Data Structures

  Data structures are defined in terms of the physical memory format of each structure. Concrete language bindings for each data structure are provided in system definition files external to this standard.

- Data Structure fields

  Record fields are referred to by using the name of the record or subrecord followed by a dot, followed by the field name; as in RECORD_NAME.SUB_RECORD.FIELD.

Part I: ALPHA-64 Calling Standard

This part of the document describes the ALPHA-64 Calling standard in it's entirety.

CHAPTER 1

INTRODUCTION

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

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

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

NOTE

In many cases significant performance gains can be realized by selective use of non-standard calls when the safety of such calls is known. Compiler writers are encouraged to make optimal use of such optimizations as appropriate.

The conventions specified in this standard are intended to fully exploit the architectural and performance advantages of the ALPHA hardware architecture and are designed to provide a leadership execution environment for applications and languages on ALPHA/VMS (phase 2), ULTRIX, and OZIX. Some of these conventions are visible to the high level language programmer, and therefore may require source level changes in high level language programs when moving them from other environments (such as VAX/ULTRIX or VAX/VMS).

To achieve source level compatibility and portability between the ALPHA-64 environment and various other environments users should not depend on the properties of this architecture except indirectly through high level language facilities that are portable across architectures.

By definition, many of the conventions described in this standard differ from other software implementation architectures. Therefore programs that depend on properties of this architecture may not be portable to other architectures.

1.1 Applicability

This standard defines the rules and conventions that govern the *native user mode run time environment* on ALPHA systems. It is applicable to all Digital Equipment Corporation products executing in native user mode on ALPHA operating systems which support this standard. This includes but is not limited to ALPHA/VMS (phase 2), ALPHA/ULTRIX and ALPHA/OZIX systems.

Specific examples of uses of this standard are:

- All externally callable interfaces in Digital-supported, standard system software
- All intermodule calls to major software components
- All external procedure calls generated by Digital language processors without the benefit of interprocedural analysis or permanent private conventions (such as those used for language support RTL routines).

1.2 Architectural Level

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

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

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

1.3 Related Documents

This calling standard is a component of the larger ALPHA Software Architecture, and depends on certain standards and conventions that are not described by this document.

Those standards, described by other documents, include:

- Object language and object file format
- Status values and message definition, formatting, and reporting
- Heap memory management and dynamic string management
- Concert Multithread Architecture
- Names and naming conventions

The above topics as well as other related topics may be found in:

- Digital Equipment Corporation ALPHA System Reference Manual, Revision 3.0, March 2, 1990
- Digital Equipment Corporation ALPHA Calling Standard ECO Process, Revision 2.0, March 16, 1990
- Digital Equipment Corporation Concert Multithread ™ Architecture Services
- Digital Equipment Corporation Extensions to MIPS R-Series Language Level Run-Time Software Architecture
- Digital Equipment Corporation VAX Procedure Calling and Condition Handling Standard, Revision 10.3
- Digital Equipment Corporation Standard 032, VAX Architecture Standard
- Digital Equipment Corporation ALPHA Object File Format Specification [TBS]
- POSIX 1003.1, IEEE Standard Portable Operating System Interface for Computer Environments - IEEE Std 1003.1-1988
- Draft Proposed American National Standard for Information Systems Programming Language C - X3J11/88-159
- Guide to Designing Portable Programming Interfaces, Version 1.0, December 1989

1.4 Definitions

The following terms are used by this standard:

- *Unpredictable*: Refers to any results of an operation which cannot be guaranteed across all implementations of this standard. These results may or may not be well defined for a single implementation but remain unpredictable with reference to this standard. All results caused by operations defined in this standard that are not explicitly specified in this standard are considered *unpredictable*. No standard conforming procedure may depend on *unpredictable* results.
- *Undefined*: Refers to operations or behavior for which there is no directing algorithm that is used across all implementations that support this standard. Such operations may or may not be well defined for a single implementation but still remain undefined with reference to this standard. The actions of undefined operations may not be required by standard conforming procedures.
- *Standard conforming procedure*: A procedure is said to be standard conforming if it adheres to all the relevant rules set forth in this document.
- *Standard call*: A standard call is defined as any transfer of control to a procedure by any means that presents the called procedure with the environment defined by this document and does not place additional restrictions, not defined by this document, on the called procedure.
- A *process* includes an address space and at least one thread of execution. Selected security and quota checks are done on a per process basis.

  An ALPHA operating system may provide multiple threads of execution within a process. An operating system which only provides a single thread of execution per process is considered to be a *special case* of a multithreaded system where the maximum number of threads per process is one.
- *Thread of execution*: sometimes referred to as simply *thread*, is the entity that is scheduled for execution on a processor. In language terms, a thread is a computational entity utilized by a program unit. Such a program unit might be a task, a procedure, a loop, or some other unit of computation.

  All threads executing within the same process share the same address space and other process context, but have unique per-thread stack and hardware context which includes program counter, processor status, stack pointer, scalar registers R0..R29, and floating registers F0..F30.

  This standard applies only to threads which execute within the context of a user mode process, and which are scheduled on one or more processors according to software priority. All subsequent uses of the term *thread* in this standard refer to such user mode process threads only.
- *Thread safe code*: refers to a property of code that has been compiled in such a way as to insure that it will execute properly when run in a threaded environment. This usually adds extra instructions to do certain run time checks as well as requiring that thread local storage be accessed in a particular fashion.
- *Natural alignment* is an attribute of certain data types that refers to the placement of the data such that the lowest addressed byte of the data has an address which is a multiple of the size of the data in bytes. Natural alignment of an aggregate data type

generally refers to an alignment such that all members of the aggregate are naturally aligned.

There are five natural alignments defined by this standard:

- 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

- *Asynchronous Software Interrupt*: refers to an asynchronous interruption of normal code flow that is caused by some software event. This interruption shares many of the properties of hardware exceptions which includes forcing some out-of-line code to execute. Only certain operating systems support this type of interrupt.
- *Exception*: refers to some exceptional condition in the current hardware and/or software state that should be noted or fixed. Its existence causes an interrupt of program flow and forces execution of out of line code. Such an event may be caused by exceptional hardware state such as arithmetic overflows, memory access control violations, etc. or by actions performed by software, such as subscript range checking, assertion checking, or asynchronous notification of one thread by another.

  While the normal control flow is interrupted by an exception, that exception is termed *active*.

  ```
  \\
  VAX/VMS documentation differentiates between the cause (exception) and effect
  (condition) when referring to an exceptional event. This document uses the
  general concept of exception to refer to the event in its entirity
  without making the cause/effect distinction.

  Most usages of exception in this document correspond with the
  VAX/VMS usage of the word condition.
  \\
  ```

- *Hardware Exception*: refers to a particular category of exceptions that directly reflect an exceptional condition in the current hardware state that should be noted or fixed by the software. Hardware exceptions may occur either synchronously or asynchronously with respect to the normal program flow.
- *Exception handler*: is a procedure which is designed to handle exceptions when they occur during the execution of a thread.
- *Signal*: is a POSIX defined concept that is used to cause out-of-line execution of code. (This term should not be confused with the VMS usage of the word which more closely equates to *exception* as used in this document)
- *Call frame*: refers to that body of information that a procedure must save to allow it to properly return to its caller. A call frame may exist on the stack or in registers. A *call frame* may optionally contain additional information required by the called procedure.
- *Procedure*: (sometimes called *simple procedure*) refers to a body of code that constitutes a particular scope which is smaller than a module and larger than a basic block.
- *Bound Procedure*: refers to a type of procedure which requires knowledge of a dynamically (at run time) determined larger enclosing scope in order to function correctly.

CHAPTER 2

BACKGROUND

This section describes various background information that served as the basis for many of the decisions made during the process of generating this standard.

2.1 Goals

The ALPHA-64 calling standard shares many goals with the VAX/VMS calling standard which include:

- The standard must be applicable to all intermodule callable interfaces in the native software system. Specifically, the standard must consider the requirements of important compiled languages including ADA, BASIC, BLISS, C, C++, COBOL, FORTRAN, PASCAL, LISP, PL/I, and calls to the operating system and library procedures. The needs of other languages that Digital may support in the future must be met by the standard or by compatible revision to it.
- The standard should not include capabilities specifically for lower level components (such as assembler routines) that cannot be invoked from the higher level languages.
- The calling program and called procedure can be written in different languages. The standard attempts to reduce the need for use of language extensions for mixed language programs.
- The standard should contribute to the writing of error free, modular, and maintainable software. Effective sharing and re-use of software modules are specific goals.
- The standard should provide the programmer with control over fixing, reporting, and flow of control when various types of exceptional conditions occur.
- The standard should provide subsystem and application writers with the ability to override system messages to provide a more suitable application oriented interface.
- The standard should add no space or time overhead to procedure calls and returns that do not establish exception handlers and should minimize time overhead for establishing handlers at the cost of increased time overhead when exceptions occur.

As well as the above goals the ALPHA calling standard has additional goals which include:

- Provide a common, compatible native user mode run time environment on ALPHA for the VMS (phase 2), ULTRIX and OZIX operating systems.
- Provide support for a multilanguage, multithreaded execution environment, including efficient, effective support for the implementation of the Digital Equipment Corporation Concert Multithread Architecture.

- 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.
- 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. This should allow a compiler to determine whether or not 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 exception handling, traceback, and debugging for lightweight procedures that do not have a stack frame.
- Make efficient and effective use of the ALPHA hardware architecture including effectively using a larger number of registers than a conventional VAX has.
- Minimize the cost of procedure calls
- Support a 64-bit address user mode environment
- Provide building block for the next 20 years of computing

2.2 Constraints

This standard was developed under the following constraints:

- The standard must be implementable on all ALPHA platforms
- The standard must be implementable by third party compiler writers.
- There is a short development cycle for the first wave of products

2.3 Tradeoffs

While the overriding goal of the ALPHA-64 flavor of the calling standard is to build a calling standard for the future, there are certain tradeoffs that have been made. These tradeoffs are documented below:

Table 2–1: ALPHA-64 Calling Standard Tradeoffs

Issue	Reason
None.	[TBS]

2.4 Important Technical Decisions

This section discusses important technical decisions that had a dramatic effect on the ALPHA-64 calling standard.

2.4.1 PC Mapping vs. FP Based Call Chain Navigation

One of the more controversial issues on the ALPHA calling standard has been which type of call chain navigation mechanism should be used for the ALPHA-64 environment flavor of the calling standard. The two choices that were considered were:

- FP based technique
- PC mapped technique

The PC mapped technique was selected for use in the ALPHA-64 environment. Below is a brief explanation of each of these methods and the rationale behind this choice.

The keys to being able to properly identify the sequence of calls that lead to any given instance of a procedure invocation is being able to identify which procedure is active at any instant in time and leaving a trail of this information behind as new procedure instances are realized.

FP Based Method

The FP method of procedure tracking is based on the requirement that at any (and *all*) point(s) in time a predefined register (called the frame pointer or FP) must contain some value that can be used to determine the current procedure (for the purposes of this discussion we will call this FP value a *procedure context value*. The contents of this register *defines* the current procedure. This implies that valid transitions of this register must only consist of one procedure context value to another procedure context value. This is accomplished by having a called procedure perform the following steps:

1. Save the current FP contents
2. Set the FP to contain the current procedure context value
3. Perform the work of the procedure (which may include other calls)
4. Restore the previous procedure context value to FP

Because the FP register must ALWAYS contain a valid procedure context value for the current procedure, the current procedure can always be determined. Furthermore, if step 1 above is required to save the previous FP contents in a well specified location then the previous procedure can also be determined, as well as the procedure before that, and the procedure before that, etc. This, then, provides the basis for one method of call chain navigation.

PC Mapped Method

The PC mapped method for call chain navigation is based on having the ability to lookup any given PC in a table that maps PC values to procedure context values. This method requires no saving or restoring of previous values since the value of the PC itself is used to determine the current procedure. Call chain navigation is done by looking up the current PC value in the table and retrieving the procedure context value. This value must in some way

provide the mechanism whereby the return address of the procedure call may be located. Once this return address is located it may be used as a PC value to look up the previous procedure context value, and so on, through the entire call chain.

Tradeoffs

Each of the above methods has as its basis one *absolute* or basic assumption. If that assumption **ever** fails then the entire mechanism fails. For the FP method the assumption is that the FP always contains a valid procedure context value. For the PC mapped method the assumption is that the PC value can always be found in a table (possibly/probably with multiple parts) of PC values or that there is an acceptable fallback strategy.

When one is designing/implementing a new system, all sorts of things must be considered. High on that list are:

- Performance
- Extensibility
- Reliability
- Supportability
- Compatibility

Each of these things relates to this issue.

Performance

Although the details of implementing either method dictate a number of implementation differences between the two, the major attraction to the PC method is that is requires no saving and restoring of the procedure context value as does the FP based method. This provides a speed advantage which in the end indicates that the PC mapped technique should be the preferred method.

Careful analysis of the instruction sequences involved indicate that the execution speed differential between the two mechanisms is normally in the order of 2 to 5 percent but can become much larger for certain categories of procedures.

Extensibility

Extensibility is one of the key areas where the advantages of the FP method appear to be greater. The key assumption for the PC mapped method is to be able to find the PC in a table. But this implies ALL PC values must be mapped in some table including:

- System bootstrap code
- Static System code
- Dynamically (re)loaded system code
- User programs
- Dynamically loaded shared images
- Transfer vectors in shared images
- Autoloading code
- Run time generated code

ANY and ALL PC values must be mapped in some table. One can see from the above list that in many cases this may be a very non-trivial exercise to accomplish. The FP method provides more control over exactly when the context is switched since the actual context switch is done at a programmed point (as opposed to the PC mapped method where it happens at the point of call). This allows the FP method to handle many of the above cases by allowing the code to execute in the context of it's caller.

Over the years many *creative* techniques have been found that involve special instruction sequences in special locations (such as putting a JSB instruction inside a structure and then executing it to pass the address of the structure). One can argue the merits of such techniques for many hours but the point here is that this type of flexibility is more difficult using a PC mapped technique. One can imagine some special systems code to do autoload of shared images that might use something like this.

Thus the FP method appears to be more flexible or extensible.

Reliability
Each of the above methods has their own set of reliability problems. In the FP method, if the FP gets invalidated somehow then all is lost. In the PC method, a jump to an unplanned location or any code that is not mapped in the PC table will cause the same effect. One must make a value judgment as to which might be more likely to occur. After considering all of the various pieces of code involved in a system it appears that both methods can be made to work reliably so this is not a determining factor.

Supportability
As long as either method works it is in some sense *supportable*, but when the PC mapped method appears to require that more pieces of a system are involved. This pieces include code such as:

- Image activation code
- Image builder (linker) tools
- Debuggers (may move code before executing it)
- System component loaders

This list is by no means exhaustive. By virtue of its simplicity and the fact that it is position independent the FP method requires much less intervention by various components of the system.

Compatibility
Compatibility is not a particularly strong point in favor of either method. To begin with, this mechanism is only visible to programmers that are dealing with assembler level code. Many people feel that the ALPHA architecture itself will help motivate the industry trend away from this level of interaction.

Summary
After taking into account all of the various aspects involved in this particular decision, the engineering tradeoffs discussed above appear to to be manageable relative to the speed advantage that can be gained by utilizing the PC mapped method. For this reason the PC mapped method has been chosen for the ALPHA-64 calling standard.

CHAPTER 3

BASIC CONSIDERATIONS

This section describes some fundamental concepts of the ALPHA-64 calling standard.

3.1 Address Representation

In the ALPHA-64 calling standard all addresses are represented as 64 bits. The only restrictions placed on what values may be used are defined by the particular operating system environment.

3.2 Procedure Representation

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

In the ALPHA-64 calling standard a simple (not bound) procedure value is defined as the address of that procedure's descriptor (see Section 4.1, Procedure Types). This provides slightly different conventions than might be used if the address of the procedure's code were used as it is in many calling standards.

A bound procedure value is defined as the address of a bound procedure descriptor, as defined in Section 4.5.4, Bound Procedure Values, that provides the necessary information for the bound procedure to be called.

In the ALPHA-64 calling standard **all** procedure values are defined to be the address of the data structure (a procedure descriptor) that describes that procedure. This design has a desired characteristic that that any procedure may be invoked by calling the address stored at offset 8 from the address represented by the procedure value.

3.3 Register Usage Conventions

This section describes the usage of the ALPHA hardware general purpose (integer) and floating point registers.

3.3.1 Integer Registers

In a standard conforming procedure the general purpose, integer registers are used as follows:

R0	Function value register. In a standard call that returns a non-floating point function result in a register, the result must be returned in this register. In a standard call, this register may be modified by the called procedure without being saved and restored.
R1	Conventional scratch register. In a standard call, this register may be modified by the called procedure without being saved and restored.
R2..R15	Conventional saved registers. If a standard-conforming procedure modifies one of these registers, it must save and restore it.
R16..R21	Argument registers. In a standard call, up to six non-floating point items of the argument list are passed in these registers. In a standard call, these registers may be modified by the called procedure without being saved and restored.
R22..R24	Conventional scratch registers. In a standard call, these registers may be modified by the called procedure without being saved and restored.
R25	AI, Argument Information register. In a standard call, this register describes the argument list (see Section 4.5.1, Call Conventions, for a detailed description). In a standard call, this register may be modified by the called procedure without being saved and restored.
R26	RA, Return Address register. In a standard call, the return address must be passed in this register. In a standard call, this register may be modified by the called procedure without being saved and restored.
R27	PV, Procedure value register. In a standard call, the procedure value of the procedure being called is passed in this register. In a standard call, this register may be modified by the called procedure without being saved and restored.
R28	Volatile scratch register. The contents of this register are always *unpredictable* after any external transfer of control either to or from a procedure. **This applies to both standard and nonstandard calls.** This register may be used by the operating system for external call fixup, autoloading and exit sequences.
R29	FP, Stack Frame Base 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.
R30	SP, the Stack Pointer. This register contains a pointer to the top of the current operating stack. Aspects of its usage and alignment are defined by the hardware architecture. Various software aspects of its usage and alignment are defined in Section 4.5.1, Call Conventions.
R31	RZ, ReadAsZero/Sink. Hardware defined: binary zero as a source operand, sink (no effect) as a result operand.

3.3.2 Floating Point Registers

In a standard conforming procedure the floating point registers are used as follows:

F0	Floating point function value register. In a standard call that returns a floating point result in a register, this register is used to return the real part of the result. In a standard call, this register may be modified by the called procedure without being saved and restored.
F1	Floating point function value register. In a standard call that returns a complex floating point result in registers, this register is used to return the imaginary part of the result. In a standard call, this register may be modified by the called procedure without being saved and restored.
F2..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, may 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 may be passed by value in these registers. In a standard call, these registers may be modified by the called procedure without being saved and restored.
F22..F30	Conventional scratch registers. In a standard call, these registers may be modified by the called procedure without being saved and restored.
F31	ReadAsZero/Sink. Hardware defined: binary zero as a source operand, sink (no effect) as a result operand.

CHAPTER 4

FLOW CONTROL

The following sections contain descriptions of various aspects of the calling standard that deal with the flow of control of a program (as opposed to data manipulation which comes later in Chapter 5).

4.1 Procedure Types

This standard defines three basic types of procedures. A compiler may chose 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
- No Frame procedure - A procedure that executes in the context of its caller

Every procedure *must* have a structure associated with it that describes which type of procedure it is as well as various other characteristics of the procedure. This structure, called a *Procedure Descriptor*, is a quadword-aligned data structure that provides basic information about a procedure. This data structure is used to interpret the call chain at any point in a thread's execution. It is normally built at compile time and is not normally accessed at run time except in support of exception processing or other rarely used code execution.

Read access to procedure descriptors is done though a procedure interface described in Appendix B, Procedure Descriptor Access Routines. This allows for future extensions to these structures in a compatible way.

The purpose of defining a procedure descriptor for a procedure, and for making that procedure descriptor accessible to the run time system is so that the invocations of that procedure are visible to and interpretable by facilities such as the debugger, the exception handling system, and the unwinder, and so that the context of the caller saved by the called procedure can be restored if an unwind occurs.

Some procedures maintain their call frame on the stack, others maintain their call frame entirely in registers (although they may use the stack). Very simple procedures do not necessarily maintain any call frame at all and simply execute in the context of their caller. The calling procedure need not distinguish these cases. The procedure descriptor for the current procedure contains a field that indicates which kind of procedure is executing.

4.1.1 Stack Frame Procedures

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" referring to the cost of storing this context in memory.

Such a procedure 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 non-intuitive ways such as:

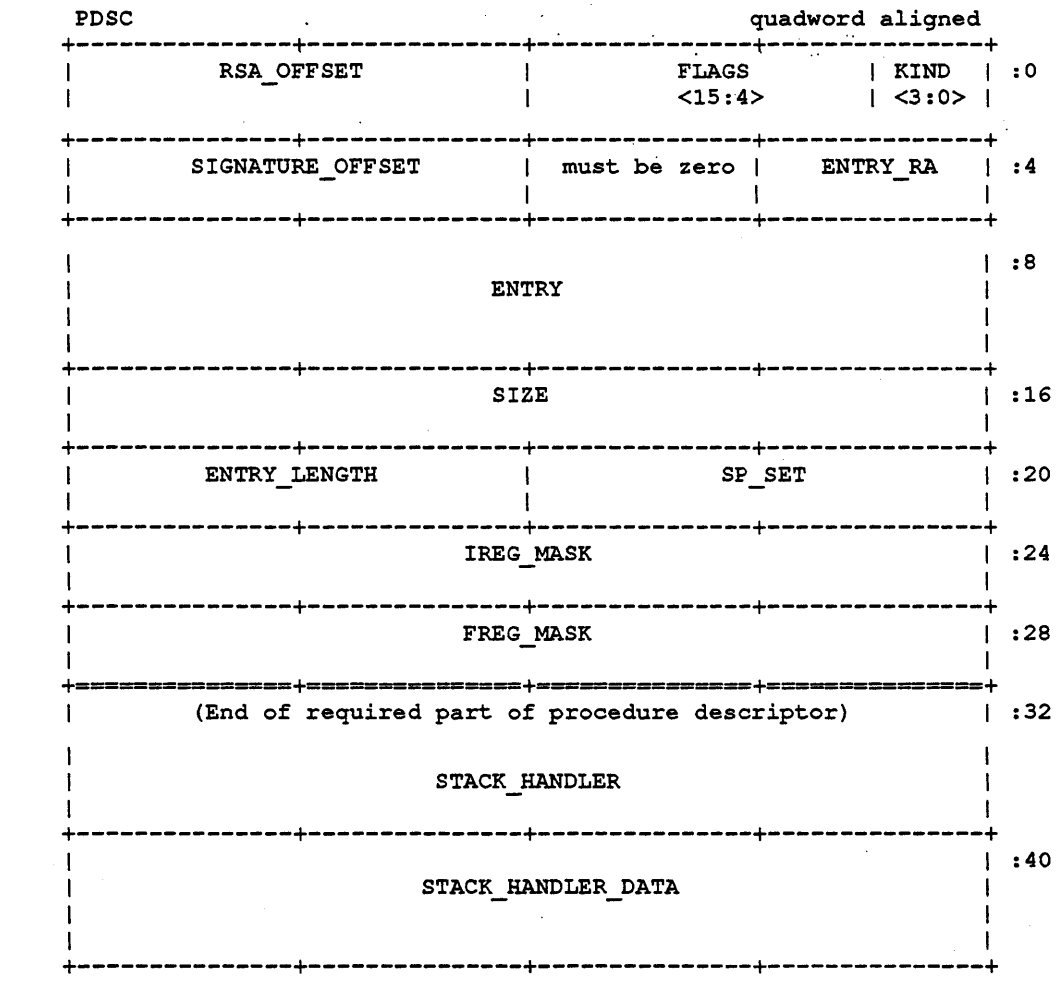
- A called routine may use the stack as a means to return certain types of function values (see Section 5.2.7, Returning Data, for details).

If any such situation exists a compiler *must* choose to use a variable frame procedure when compiling the caller so that an unwind operation can be done correctly.

Procedure Descriptor For Procedures With a Stack Frame

Procedure descriptors for procedures with stack frames are defined as follows:

Figure 4–1: Stack Frame Procedure Descriptor Format



PDSC.KIND = 1 (denoted by PDSC_KIND_PC_STACK) specifies a procedure descriptor for a procedure with a stack frame. The base of the stack frame for this type of procedure is always pointed to by either SP or FP as indicated by PDSC.FLAGS.BASE_REG_IS_FP.

PDSC.FLAGS is a vector of flag bits defined as follows:

- PDSC.FLAGS<0> is denoted by HANDLER_VALID.

  If HANDLER_VALID is 1, then PDSC.STACK_HANDLER must specify an exception handler.

- PDSC.FLAGS<1> is denoted by HANDLER_REINVOKABLE.

  HANDLER_REINVOKABLE is 1 if the exception handler is a reinvokable handler, that is, one that can be reinvoked should another exception occur while the handler is already active. If HANDLER_REINVOKABLE is 0 then the exception handler is not reinvokable.

HANDLER_REINVOKABLE must be 0 unless PDSC.FLAGS.HANDLER_VALID is 1.

RATIONALE:

This flag was added to correct a specific behavior that became apparent while implementing run time support using the VAX/VMS exception handling. If an exception arises while another exception handler is already active, VAX/VMS skips all handlers between the active frame and the establisher of the exception handler in which the new exception occurred. BASIC and PL/I have both had to code around this problem. Use of this flag would allow the exception dispatch code to not skip any exception handler that had this bit set.

- PDSC.FLAGS<2> is denoted by HANDLER_DATA_VALID.

  If HANDLER_DATA_VALID is 1, then PDSC.STACK_HANDLER must specify an exception handler, PDSC.FLAGS.HANDLER_VALID must be 1, and PDSC.STACK_HANDLER_DATA must exist. If all of these criteria are met the address of PDSC.STACK_HANDLER_DATA will be passed to the exception handler as defined in Section 6.1, Exception Handling.

- PDSC.FLAGS<3> is denoted by BASE_REG_IS_FP.

  If BASE_REG_IS_FP is 0, then the SP is used as the base register to which PDSC.SIZE is added during an unwind. A procedure with this flag set to 0 has a fixed amount of stack storage specified by PDSC.SIZE, all of which is allocated in the procedure entry sequence, and SP is modified by this procedure only in the entry and exit code sequences. In this case R29 (FP) is simply another register and has no special significance.

  If BASE_REG_IS_FP is 1, then R29 (FP) is used as the frame base pointer. A procedure with this flag set to 1 has a fixed amount of stack storage specified by PDSC.SIZE, and may have a variable amount of stack storage allocated by modifying SP in the body of the procedure.

- PDSC.FLAGS<4> is denoted by REI_RETURN.

  REI_RETURN is 1 if the procedure expects the stack at entry to be set up so that an REI instruction will correctly return from the procedure. If this bit is set the contents of both the PDSC.ENTRY_RA field and the RSA.SAVED_RETURN field in the register save area are *unpredictable* and the return address is found on the stack. (See Figure 4–4, Register Save Area Layout, for details.)

- PDSC.FLAGS<5> is denoted by STACK_RETURN_VALUE.

  STACK_RETURN_VALUE is 1 if the procedure does not reset the stack pointer to its value at procedure entry. This is the case when the function value is returned on the stack.

- PDSC.FLAGS<6> *must* be 0.

- PDSC.FLAGS<7> is denoted by NO_JACKET.

  NO_JACKET must always be set to 1 for compiled code.

  RATIONALE:

  The PDSC.FLAGS.NO_JACKET and PDSC.FLAGS.NATIVE flag bits are carefully designed to align with bits 12 and 13 of a VAX procedure entry mask which must always be zero.

- PDSC.FLAGS<8> is denoted by NATIVE.

NATIVE must always be set to 1 for compiled code.

- PDSC.FLAGS<11:9> *must* be 0.

PDSC.RSA_OFFSET is the signed difference in bytes between the stack frame base (SP or FP as indicated by PDSC.FLAGS.BASE_REG_IS_FP) and the register save area (see layout below). PDSC.RSA_OFFSET must be a multiple of 8 such that (PDSC.RSA_OFFSET) added to the contents of SP or FP, as indicated by PDSC.FLAGS.BASE_REG_IS_FP, yields a quadword-aligned address.

PDSC.ENTRY_RA is the number of the register in which the return address is passed to this procedure (26 for a standard call).

RATIONALE:

ENTRY_RA is used, as opposed to SAVE_RA, to allow the return address to be passed in a non-standard register. If it were not present and an exception happened in a procedure prolog there would be no way to determine where to find the return address (since SAVE_RA is not valid until the prolog is completed).

PDSC.SIGNATURE_OFFSET is a 16-bit signed byte offset from the start of the procedure descriptor. This offset, if non-zero, designates the start of the procedure signature block. A zero in this field indicates that no signature information is present. (See Section 4.2, Procedure Signatures, for details of the procedure signature block.)

PDSC.ENTRY is the absolute address of the first instruction of the entry code sequence for the procedure.

PDSC.SIZE is the unsigned size in bytes of the fixed portion of the stack frame for this procedure. The value of SP at entry to this procedure can be calculated by adding PDSC.SIZE to the value SP or FP, as indicated by PDSC.FLAGS.BASE_REG_IS_FP. PDSC.SIZE cannot be 0 for a stack frame type of procedure since the stack frame must include space for the register save area.

PDSC.SIZE *must* be an appropriate multiple to maintain the minimum stack alignment required by the ALPHA hardware architecture. Furthermore, in almost all cases it will be a multiple of 16 to maintain proper stack alignment during a call as defined in Section 4.5.1, Call Conventions.

Various combinations of PDSC.FLAGS.BASE_REG_IS_FP and PDSC.SIZE can be used as follows:

- When PDSC.FLAGS.BASE_REG_IS_FP is 0 and PDSC.SIZE is zero, then the procedure utilizes no stack storage and SP contains the value of SP at entry to the procedure. (Procedure must be a register frame procedure)
- When PDSC.FLAGS.BASE_REG_IS_FP is 0 and PDSC.SIZE is non-zero, then the procedure has a fixed amount of stack storage specified by PDSC.SIZE, all of which is allocated in the procedure entry sequence, and SP is modified by this procedure only in the entry and exit code sequences.
- When PDSC.FLAGS.BASE_REG_IS_FP is 1 and PDSC.SIZE is non-zero, then the procedure has a fixed amount of stack storage specified by PDSC.SIZE, and may have a variable amount of stack storage allocated by modifying SP in the body of the procedure. (Procedure must be a stack frame procedure)

- The combination when PDSC.FLAGS.BASE_REG_IS_FP is 1 and PDSC.SIZE is zero is illegal since it violates the rules for R29 (FP) usage which say that it must be saved (on the stack) and restored.

PDSC.SP_SET is the unsigned offset of the one *and only one* instruction in the procedure prologue which modifies the stack pointer.

PDSC.ENTRY_LENGTH is the unsigned offset in bytes from the entry point of the first instruction in the procedure code segment following the procedure prologue.

PDSC.IREG_MASK 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 R0. (Bits 31, 30, and 28 of this mask should never be set since R31 is the integer Read-As-Zero register, R30 is the hardware SP, and R28 is always assumed to be destroyed during a procedure call or return.)

PDSC.FREG_MASK is a bit vector (0..31) specifying the floating point registers that are saved in 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 since it corresponds to the floating point Read-As-Zero register.)

PDSC.STACK_HANDLER is a signed self-relative pointer to the procedure descriptor for a run time static exception handling procedure. This part of the procedure descriptor is optional. It MUST be supplied if either PDSC.FLAGS.HANDLER_VALID is 1 or PDSC.FLAGS.HANDLER_DATA_VALID is 1 (which requires that PDSC.FLAGS.HANDLER_VALID be 1).

If PDSC.FLAGS.HANDLER_VALID is 0, then the contents or existence of PDSC.STACK_HANDLER is *unpredictable.*

PDSC.STACK_HANDLER_DATA is a quadword, the address of which is passed to the exception handler as context. This is an optional quadword and need only be supplied if PDSC.FLAGS.HANDLER_DATA_VALID is 1.

If PDSC.FLAGS.HANDLER_DATA_VALID is 0, then the contents or existence of PDSC.STACK HANDLER_DATA is *unpredictable.*

RATIONALE:

Note that the handler data cell is intended for use by compilers to help implement the appropriate language semantics. The calling standard does not provide any direct means by which user level code may cause data to be stored in this location. Note also that because the address of the handler data cell is passed instead of the data itself, the data may be a self relative pointer.

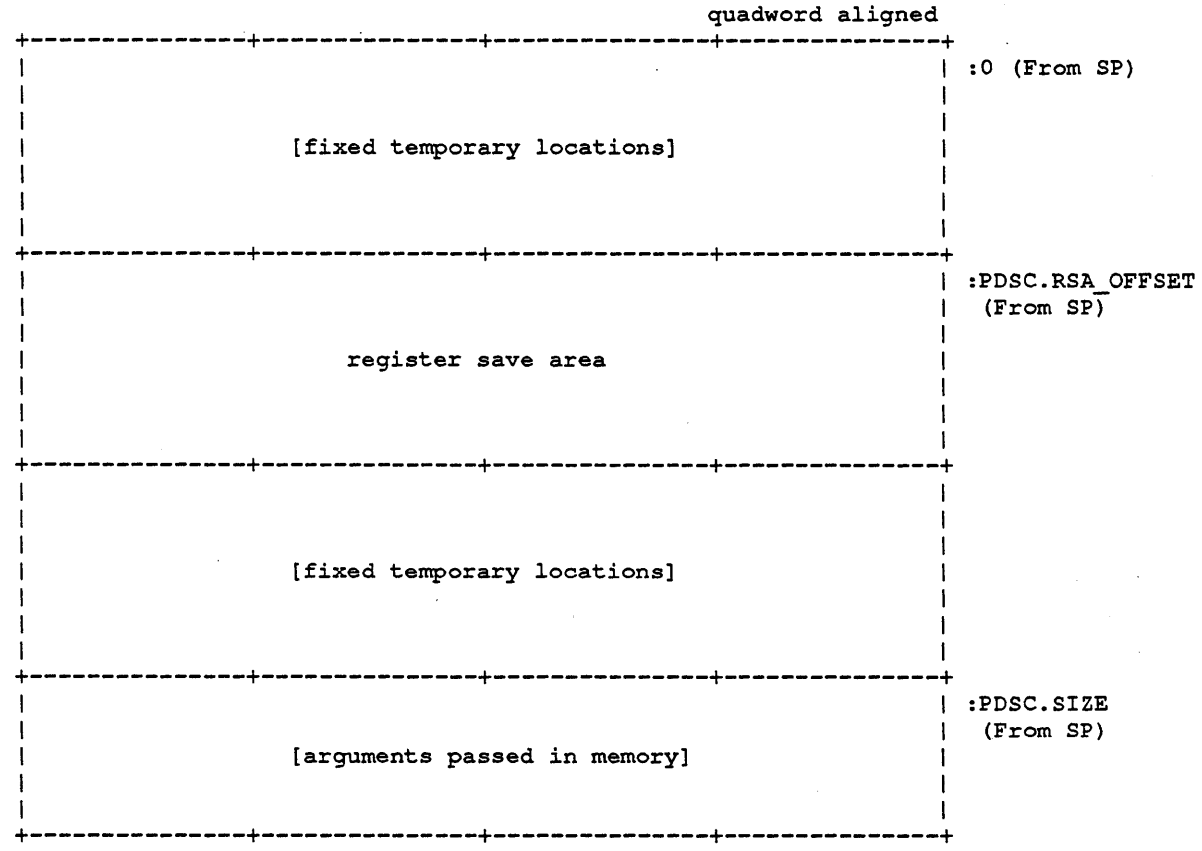
Stack Frame Format

Even though the exact contents of a stack frame are determined by the compiler there are certain properties common to all stack frames. The two basic flavors of stack frames are described below.

The following figure illustrates the format of the stack frame for a procedure with a fixed amount of stack which uses the SP as the stack base register (i.e. PDSC.FLAGS.BASE_REG_IS_FP is 0). In this case, R29 is simply another register and has no special significance.

Some parts of the stack frame are optional and occur only as required by the particular procedure. Brackets surrounding a field's name indicate the field is optional.

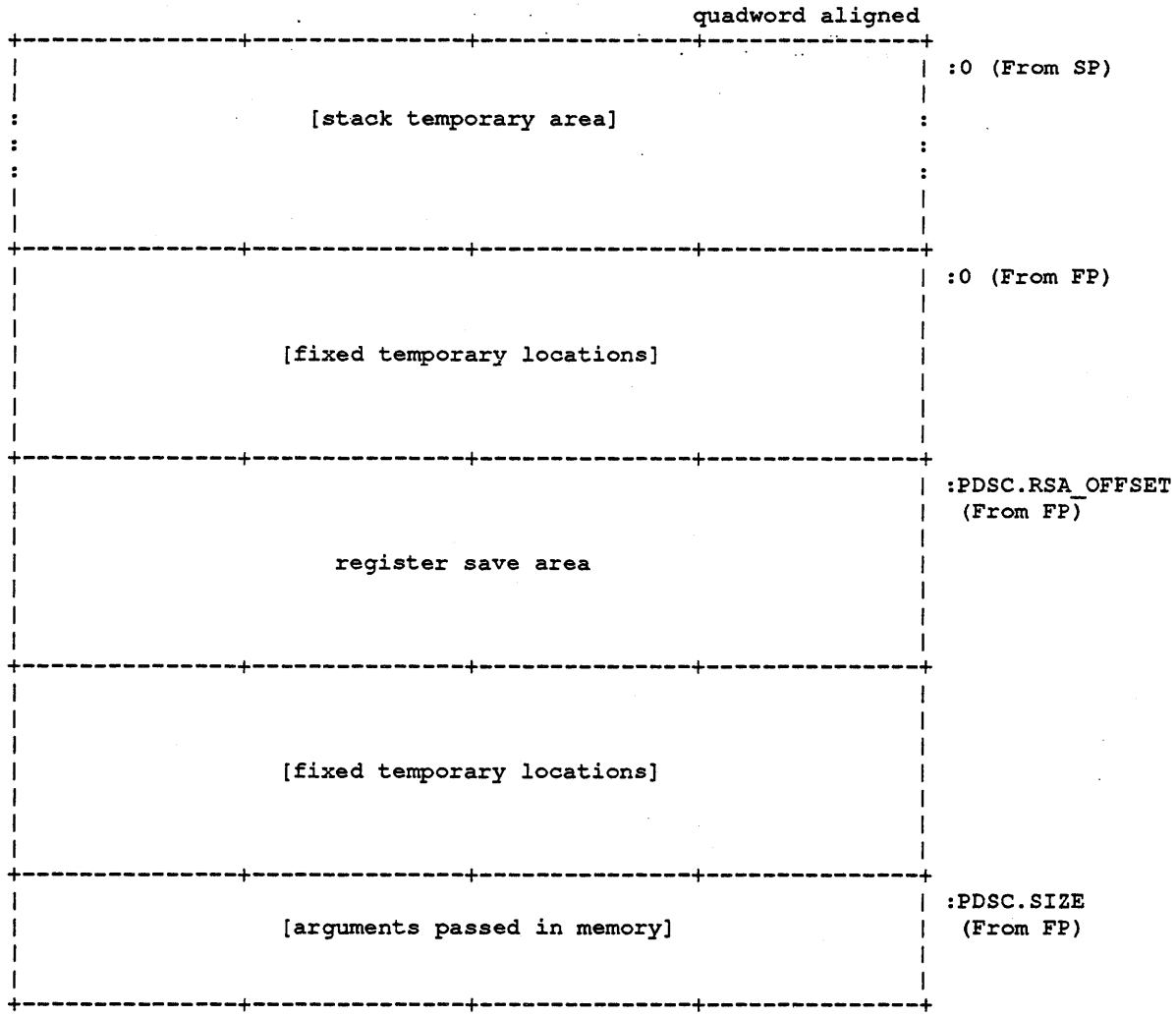
Figure 4–2: Fixed Size Stack Frame Format



The following figure illustrates the format of the stack frame for procedures with a varying amount of stack which use the FP as the stack base register (i.e. PDSC.FLAGS.BASE_ REG_IS_FP is 1).

Some parts of the stack frame are optional and occur only as required by the particular procedure. Brackets surrounding a field's name indicate the field is optional.

Figure 4–3: Variable Size Stack Frame Format



In either case the portion of the stack frame designated by PDSC.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. This may include, for example, register spill area, language-specific exception handling context (such as language dynamic exception handling information), fixed temporaries, etc.

If a compiler chooses, the fixed temporary locations adjacent to the area pointed to by the frame base register plus PDSC.SIZE can be used for a special purpose termed 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 may also be used to store arguments that are passed in registers if an address for such a argument must be generated. Generally, 6 * 8 bytes of stack storage will be allocated by the called procedure for this purpose.

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

Register save area is a set of consecutive quadwords in which registers saved and restored by the current procedure are stored. The register save area begins at the location pointed to by the frame base register (as indicated by PDSC.FLAGS.BASE_REG_IS_FP) plus the contents of PDSC.RSA_OFFSET, which must yield 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.IREG_MASK and PDSC.FREG_MASK. The details of how to lay out and populate the register save area are described below.

A compiler may use the **stack temporary area** for fixed local variables, such as constant-sized data items and program state, as well as for dynamically sized local variables. The stack temporary area may also be used for dynamically sized items with a limited lifetime, for example, a dynamically sized function result or string concatenation that can't 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 the base to reclaim storage used by temporaries.

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

Register Save Area

The layout of the frame of a stack frame procedure contains a substructure called the *Register Save Area*. This area is defined and populated as described below.

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

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

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

NOTE

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

RATIONALE:

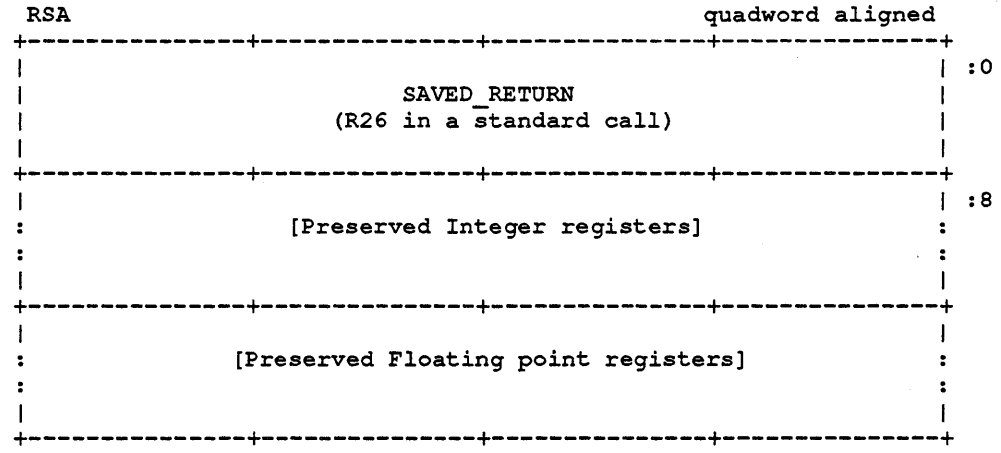
The above note is so that an unwind routine can properly restore the floating point registers without the type information.

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

If, however, a non-standard call is made that requires the return address register to be saved and restored then it must be stored in **both** the location at offset 0 in the register save area and at the appropriate location within the variable part of the save area. In addition the appropriate bit of PDSC.IREG_MASK must be 1.

The register save area layout is defined as:

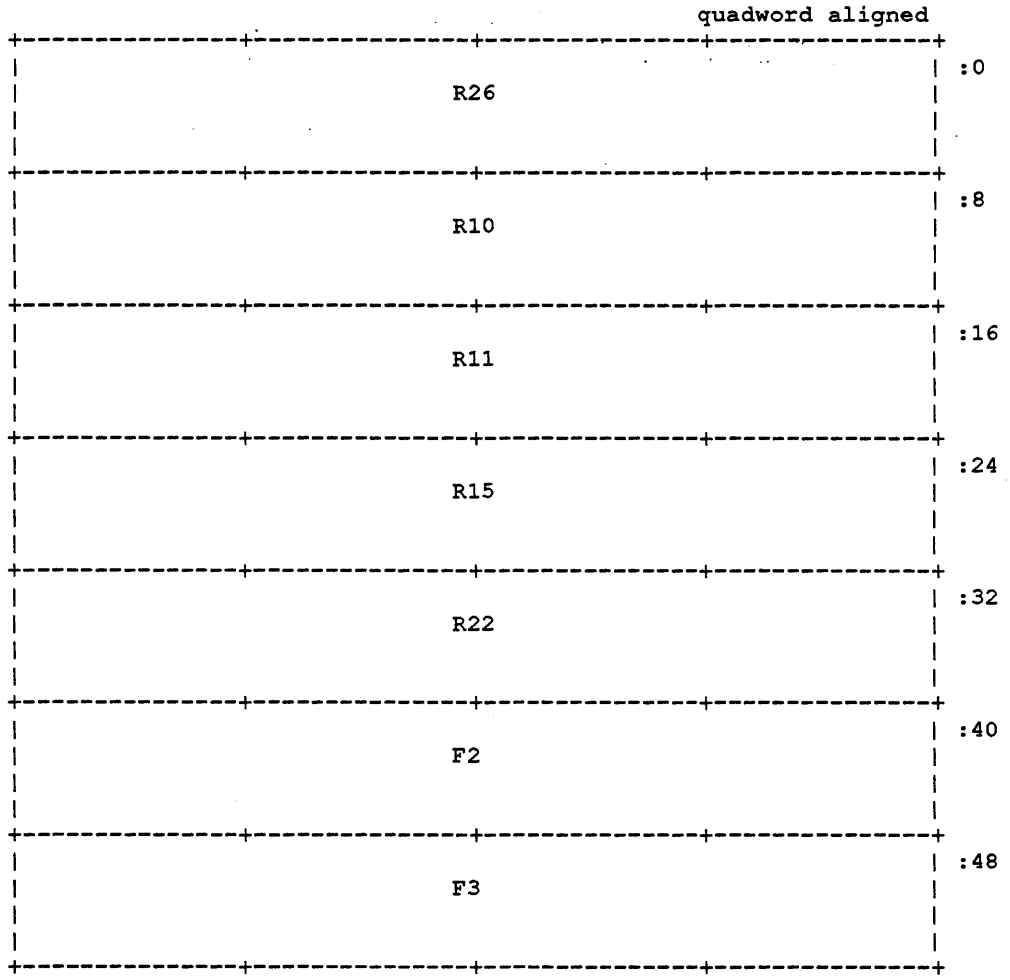
Figure 4–4: Register Save Area Layout



RSA.SAVED_RETURN is the contents of the return address register.

For example, if registers R10, R11, R15, R22, F2, and F3 were saved by a standard procedure, the IREG_MASK value would be 00008C00 (hex) and the FREG_MASK would be 0000000C (hex). The register save area for such a procedure would be packed as follows:

Figure 4–5: Register Save Example



4.1.2 Register Frame Procedure

A *Register Frame procedure* does not maintain a call frame on the stack and must therefore save its caller's context in registers. This type of procedure is sometimes referred to as a "lightweight procedure" referring to the relatively fast way of saving 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 call to any other procedure.

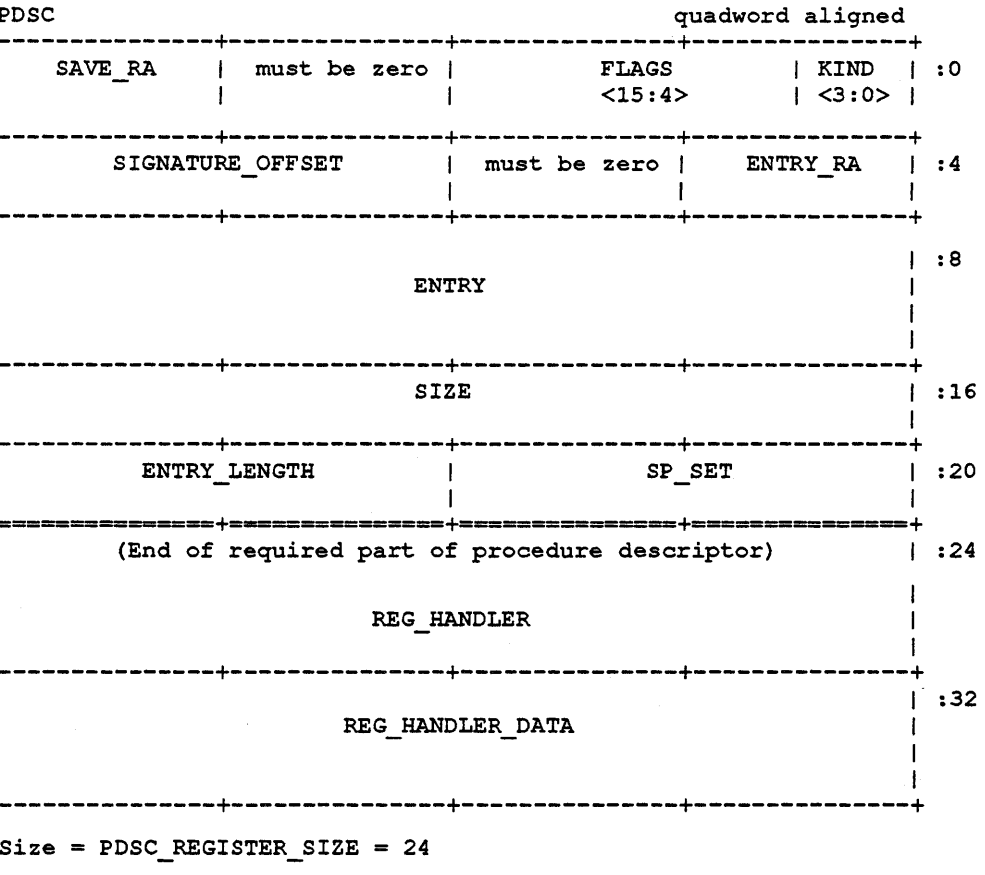
A procedure with a register frame can have an exception handler and can handle exceptions in the normal way. Such a procedure can also allocate local stack storage in the normal way, although it will not necessarily do so.

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

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

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

Procedure Descriptor For Procedures With a Register Frame

Procedure descriptors for procedures with register frames are defined as follows:

Figure 4–6: Register Frame Procedure Descriptor Format



PDSC.KIND = 2 (denoted by PDSC_KIND_PC_REGISTER) specifies a procedure descriptor for a procedure with a register frame.

PDSC.FLAGS is a vector of flag bits defined as follows:

- PDSC.FLAGS<0> is denoted by HANDLER_VALID.

If HANDLER_VALID is 1, then PDSC.REG_HANDLER must specify an exception handler.

- PDSC.FLAGS<1> is denoted by HANDLER_REINVOKABLE.

  HANDLER_REINVOKABLE is 1 if the exception handler is a reinvokable handler, that is, one that can be reinvoked should another exception occur while the handler is already active. If HANDLER_REINVOKABLE is 0 then the exception handler is not reinvokable.

  HANDLER_REINVOKABLE must be 0 unless PDSC.FLAGS.HANDLER_VALID is 1.

- PDSC.FLAGS<2> is denoted by HANDLER_DATA_VALID.

  If HANDLER_DATA_VALID is 1, then PDSC.REG_HANDLER must specify an exception handler, PDSC.FLAGS.HANDLER_VALID must be 1, and PDSC.REG_HANDLER_DATA must exist. If all of these criteria are met the address of PDSC.REG_HANDLER_DATA will be passed to the exception handler in as defined in Section 6.1, Exception Handling.

- PDSC.FLAGS<3> is denoted by BASE_REG_IS_FP.

  BASE_REG_IS_FP must always be 0 for a register frame procedure since setting it to 1 would imply that the FP was used, but that could not be correct since the FP must be saved and restored across a call and register frame procedures cannot save registers.

- PDSC.FLAGS<4> is denoted by REI_RETURN.

  REI_RETURN is 1 if the procedure expects the stack at entry to be set up so that an REI instruction will correctly return from the procedure. If this bit is set the contents of both the PDSC.ENTRY_RA and the PDSC.SAVE_RA fields are *unpredictable* and the return address is found on the stack.

- PDSC.FLAGS<5> is denoted by STACK_RETURN_VALUE.

  STACK_RETURN_VALUE is 1 if the procedure does not reset the stack pointer to its value at procedure entry. This is the case when the function value is returned on the stack.

- PDSC.FLAGS<6> *must* be 0.

- PDSC.FLAGS<7> is denoted by NO_JACKET.

  NO_JACKET must always be set to 1 for compiled code.

- PDSC.FLAGS<8> is denoted by NATIVE.

  NATIVE must always be set to 1 for compiled code.

- PDSC.FLAGS<11:9> *must* be 0.

PDSC.SAVE_RA is the number of the register in which the return address is maintained during the body of the procedure. If this procedure uses the standard call conventions and does not modify R26, then both PDSC.ENTRY_RA and PDSC.SAVE_RA will specify R26.

In a standard procedure, PDSC.SAVE_RA must specify a scratch register so as not to violate the rules for procedure entry code as specified in Section 4.5.5, Entry and Exit Code Sequences.

PDSC.ENTRY_RA is the number of the register in which the return address is passed to this procedure (26 for a standard call).

PDSC.SIGNATURE_OFFSET is a 16-bit signed byte offset from the start of the procedure descriptor. This offset, if non-zero, designates the start of the procedure signature block. A zero in this field indicates that no signature information is present. (See Section 4.2, Procedure Signatures, for details of the procedure signature block.)

PDSC.ENTRY is the absolute address of the first instruction of the entry code sequence for the procedure.

PDSC.SIZE is the unsigned size in bytes of the fixed portion of the stack frame for this procedure. The value of SP at entry to this procedure can be calculated by adding PDSC.SIZE to the value SP. PDSC.SIZE is 0 if the procedure uses no stack.

PDSC.SIZE *must* be an appropriate multiple to maintain the minimum stack alignment required by the ALPHA hardware architecture. Furthermore, in almost all cases it will be a multiple of 16 to maintain proper stack alignment during a call as defined in Section 4.5.1, Call Conventions.

PDSC.SP_SET is the unsigned offset of the one *and only one* (if any) instruction in the procedure prologue which modifies the stack pointer.

If PDSC.SIZE = 0, then the contents of PDSC.SP_SET are *unpredictable.*

PDSC.ENTRY_LENGTH is the unsigned offset in bytes from the entry point of the first instruction in the procedure code segment following the procedure prologue.

PDSC.REG_HANDLER is a signed self-relative pointer to the procedure descriptor for a run time static exception handling procedure. This part of the procedure descriptor is optional. It MUST be supplied if either PDSC.FLAGS.HANDLER_VALID is 1 or PDSC.FLAGS.HANDLER_DATA_VALID is 1 (which requires that PDSC.FLAGS.HANDLER_VALID be 1).

If PDSC.FLAGS.HANDLER_VALID is 0, then the contents or existence of PDSC.REG_HANDLER is *unpredictable.*

PDSC.REG_HANDLER_DATA is a quadword, the address of which is passed to the exception handler as context. This is an optional quadword and need only be supplied if PDSC.FLAGS.HANDLER_DATA_VALID is 1.

If PDSC.FLAGS.HANDLER_DATA_VALID is 0, then the contents or existence of PDSC.REG_HANDLER_DATA is *unpredictable.*

RATIONALE:

Because the address of the handler data cell is passed instead of the data itself, the data may be a self relative pointer.

4.1.3 No Frame Procedure

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

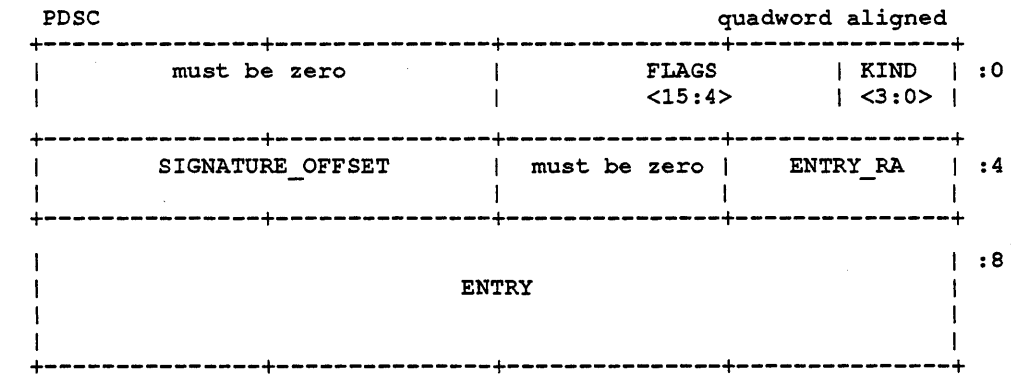
For example, if a procedure does not establish an exception handler or save and restore registers, and does not extend the stack, then that procedure may not need to establish a context. Likewise, if that procedure does extend the stack, it may still not need to establish a context if the immediate caller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

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

As with the other procedure types defined above, the choice to establish a context or not is up to the called procedure. By defining a null procedure descriptor format the same invocation code sequence may be used by the caller for all procedure types.

Procedure descriptors for procedures with no frame are defined as follows.

Figure 4–7: Null Frame Procedure Descriptor Format



PDSC.KIND = 8 (denoted by PDSC_KIND_NULL) specifies a procedure descriptor for a procedure with no frame.

```
\\
For the FP based environment of the ALPHA-32 calling standard there
are two values of PDSC.KIND (0 and 8) that must never be
pointed to by a FP. This is because the algorithm for determining if
the contents of FP points to a procedure descriptor directly or
indirectly through a stack location is based on determining if the
quadword which is pointed to has the low order 3 bits clear. Thus PDSC.KIND
values of 0 or 8 fulfill this criteria when it would not be valid.

Bound procedure descriptors and null procedure descriptors are formatted
identically for both the ALPHA-32 and the ALPHA-64 flavor of the calling
standard. Since neither of these types of procedure descriptors can
ever be pointed to by an FP they are assigned these two special
values of PDSC.KIND, thus solving the problem.
\\
```

PDSC.FLAGS is a vector of flag bits defined as follows.

- PDSC.FLAGS<3:0> *must* be 0.
- PDSC.FLAGS<4> is denoted by REI_RETURN.

REI_RETURN is 1 if the procedure expects the stack at entry to be set up so that an REI instruction will correctly return from the procedure. If this bit is set the contents of the PDSC.ENTRY_RA field are *unpredictable* and the return address is found on the stack.

- PDSC.FLAGS<5> is denoted by STACK_RETURN_VALUE.

  STACK_RETURN_VALUE is 1 if the procedure does not reset the stack pointer to its value at procedure entry. This is the case when the function value is returned on the stack.

- PDSC.FLAGS<6> *must* be 0.
- PDSC.FLAGS<7> is denoted by NO_JACKET.

  NO_JACKET must always be set to 1 for compiled code.

- PDSC.FLAGS<8> is denoted by NATIVE.

  NATIVE must always be set to 1 for compiled code.

- PDSC.FLAGS<11:9> *must* be 0.

PDSC.ENTRY_RA is the number of the register in which the return address is passed to this procedure. Since this type of procedure never establishes itself as the *current* procedure, the return address must remain in this register during the entire time the procedure is active.

PDSC.SIGNATURE_OFFSET is a 16-bit signed byte offset from the start of the procedure descriptor. This offset, if non-zero, designates the start of the procedure signature block. A zero in this field indicates that no signature information is present. (See Section 4.2, Procedure Signatures, for details of the procedure signature block.)

PDSC.ENTRY is the absolute address of the first instruction of the entry code sequence for the procedure.

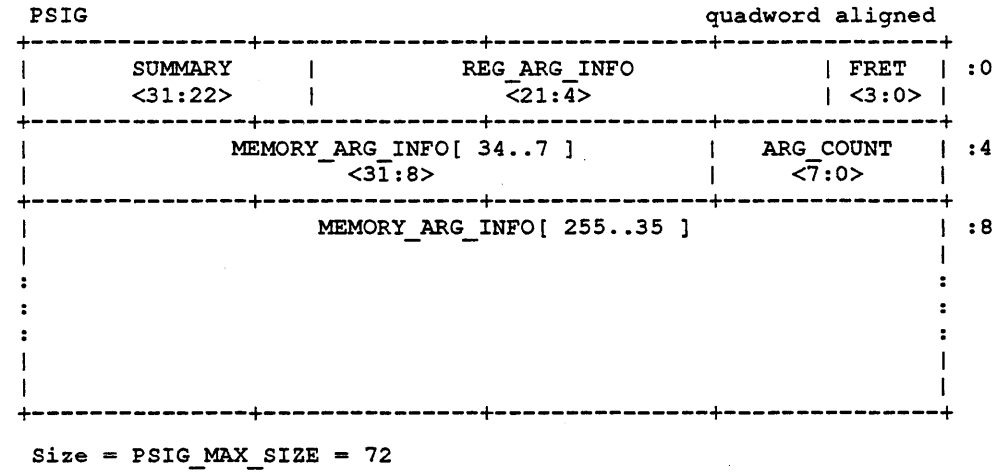
4.2 Procedure Signatures

remove?

```
\\
Current plans indicate that signature information will only be utilized in the
ALPHA-32 flavor of the calling standard (to enhance interoperability between
translated and native code). Possible uses have also been identified for the
ALPHA-64 flavor of the standard although no implementation plans have been
finalized in this area.
\\
```

As a means of enhancing certain aspects of program compatibility compilers may generate optional information describing a procedure. This auxillary information is called *signature information*. If a procedure is compiled with signature information, PDSC.SIGNATURE_OFFSET contains a byte offset from the procedure descriptor to the start of a procedure signature block. The procedure signature block is defined as follows:

Figure 4–8: Procedure Signature Block Format



PSIG.FUNC_RETURN is a 4-bit field the describes which registers will be used for the function value return (if there is one) and what format will be used for those registers. This field is encoded as follows:

Table 4–1: Function Return Signature Encodings

Name	Value	Meaning
FR_I32	0	32-bit sign extended to 64-bit result in R0
		or - No function result provided
		or - First parameter mechanism used
FR_I64	1	64-bit result in R0
FR_U32	2	32-bit unsigned result (zero extended) in R0
FR_D64	3	64-bit result with low 32 bits in sign extended R0 and high 32 bits sign extended in R1
FR_FF	4	F floating result in F0
FR_FDG	5	D or G floating result in F0
FR_FS	6	S floating result in F0
FR_FT	7	T floating result in F0
	8-11	Reserved for future use
FR_FFC	12	F floating complex result in F0 and F1
FR_FDGC	13	D or G floating complex result in F0 and F1
FR_FSC	14	S floating complex result in F0 and F1
FR_FTC	15	T floating complex result in F0 and F1

PSIG.REG_ARG_INFO<21:4> is an 18-bit bitvector consisting of 6 groups of 3 bits corresponding to the 6 arguments that may be passed in registers. These groups describe how each of the first six arguments are expected to be passed in registers with the first group (bits <6:4>) describing the first argument. Each group is encoded as follows:

Table 4–2: Register Argument Signature Encodings

Name	Value	Meaning
RA_I32	0	32-bit sign extended to 64-bit argument passed in an integer register
		or - Argument is not present
		or - Argument slot used by previous argument
RA_I64	1	64-bit argument passed in an integer register
RA_U32	2	32-bit unsigned argument zero extended to 64 bits, passed in an integer register
	3	Reserved for future use
RA_FF	4	F floating argument passed in a floating register
RA_FDG	5	D or G floating argument passed in a floating register
RA_FS	6	S floating argument passed in a floating register
RA_FT	7	T floating argument passed in a floating register

PSIG.SUMMARY<31:22> is a 10-bit field defined as follows:

Table 4–3: Signature Summary

Name	Bit	Meaning
SU_ASUM	0-1	Summary of arguments 7 through PSIG.ARG_COUNT 00 = All arguments are 64-bit or not used 01 = All arguments are 32-bit sign extended 10 = Reserved for call signature 11 = Other (Not 00 or 01)
SU_VLIST	2	VAX style argument list expected
	3-9	Reserved for future use - Must be zero

PSIG.ARG_COUNT<7:0> is an unsigned byte that represents the number of 64-bit argument items that are described in the argument signature information. This count includes the first six arguments.

PSIG.MEMORY_ARG_INFO is an array of two-bit values which describe each of arguments 7 through PSIG.ARG_COUNT. These bits are defined as follows:

Table 4–4: Memory Argument Signature Encodings

Name	Value	Meaning
MA_I64	0	64-bit argument
	1	Reserved
MA_I32	2	32-bit sign extended argument
MA_U32	3	32-bit zero extended argument

PSIG.MEMORY_ARG_INFO data is only defined for the arguments described by PSIG.ARG_COUNT.

4.3 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 as defined in Appendix B, Procedure Descriptor Access Routines.

4.4 Procedure Call Chain

4.4.1 Current Procedure

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

In order to completely determine the current execution context not only must the currently executing procedure be determined but also which instance of that procedure. This context of the current procedure together with a specific instance of that procedure invocation is called the *current procedure invocation* (which is more often shortened to *current procedure*). At any point in the execution of a thread there is always exactly one procedure that is considered to be the *current procedure*.

In the ALPHA-64 calling standard the value in the PC is used to indicate the *current procedure* by means of a lookup table.

To accomplish this **all** possible PC values must be mapped, either explicitly or implicitly, in a table of PC values. This table is composed of a set of triples; PC starting and ending values defining a range, and for that range a pointer to the appropriate procedure descriptor. Given a PC value at any point in time, that PC value can be interpreted to find the procedure descriptor for the current procedure by doing a lookup in this table. (Details of the table lookup scheme can be found in Appendix C.) Details of the runtime interface to the lookup table are found below.

The following system supplied routine may be used to obtain a procedure value (the address of the procedure descriptor) that corresponds with any given PC value within the current address space:

Get Procedure Value

LIB_GET_PROC_VALUE(PC_VALUE)

Arguments:

PC_VALUE	A PC value in the current address space for which the procedure value is to be returned If zero, indicates the value should be returned for the routine making the call

Function Value:

PROC_VALUE	The procedure value for the procedure containing the requested PC If the return value is PROC_VALUE_NULL, then the PC is not currently mapped

4.4.2 The Call Chain

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

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

4.4.3 Procedure Call Tracing

There are three basic facilities needed to support procedure call tracing. They are:

1. A mechanism to refer to a given procedure invocation
2. A mechanism to provide the context of a procedure invocation
3. A mechanism to traverse the procedure call chain

These facilities are described below.

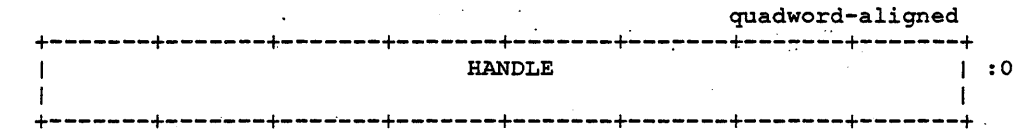
Data structures

This section describes the data structures used to provide the above mentioned mechanisms.

Referring to a Procedure Invocation

A given procedure invocation can be referred to by use of a *procedure invocation handle*. This handle is defined as follows:

Figure 4–9: Procedure Invocation Handle Format

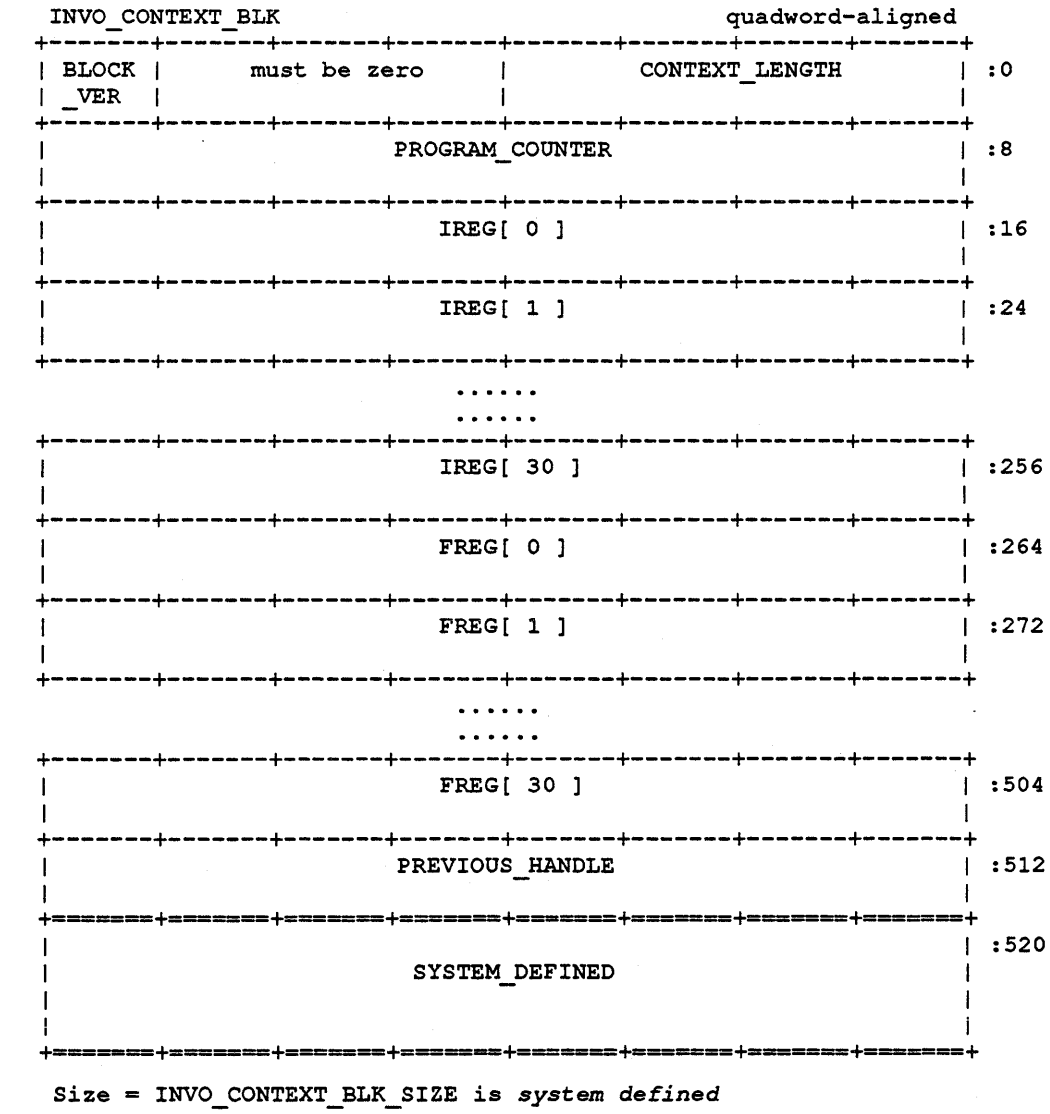


HANDLE is a 64-bit quantity which may be used to refer to a specific procedure invocation at run time. (See Section D.2.1, Invocation Handles, for how this handle might be constructed.)

Invocation Context Block

The context of a given procedure invocation is provided through the use of a data structure called an *invocation context block*. This structure is defined as follows:

Figure 4–10: Invocation Context Block Format



INVO_CONTEXT_BLK.CONTEXT_LENGTH is an unsigned count of the total length in bytes of the context block; this represents the sum of the lengths of the standard-defined portion and the system-defined portion.

INVO_CONTEXT_BLK.BLOCK_VERSION is a byte that indicates the context block version. This value would change if the format of this structure ever changes. For now, it is always set to the value 1.

INVO_CONTEXT_BLK.PROGRAM_COUNTER is the current value of the program counter in the procedure. For interrupted procedures, this is the same as the continuation program counter; for active procedures, this is the return address back into that procedure.

INVO_CONTEXT_BLK.IREG[] and FREG[] are the current values of the integer and floating point registers in the procedure.

INVO_CONTEXT_BLK.PREVIOUS_HANDLE contains the invocation handle for caller of the procedure whose context information is supplied.

INVO_CONTEXT_BLK.SYSTEM_DEFINED contains procedure context information defined by the host environment. These locations are *not* defined by this standard.

```
\\
The system defined section is to allow host environments to collect and keep in
one place all the information that needs to be collected and kept for
interrupted and active procedures.

For example, the sigcontext structure is used for this purpose in
ULTRIX/RISC. It contains all the information in this invocation context block,
plus more. Any ULTRIX implementation is likely to want to keep the additional
sorts of ULTRIX-specific information that is usually found in a
sigcontext structure in the invocation context block. This
system-defined area is provided to allow the invocation context block to be
used as the sigcontext structure.
\\
```

Referencing a Procedure

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

LIB_GET_INVO_HANDLE(INVO_CONTEXT_BLK_ADDR)

Arguments:

INVO_CONTEXT_BLK_ADDR	This is the address of a filled in invocation context block. If this address is passed as 0, the call is assumed to be a request for the invocation handle of the procedure making the call.

Function Value:

INVOCATION_HANDLE	This is an invocation handle of the invocation context that was passed. If the returned value is INVO_HANDLE_NULL, the invocation context that was passed was invalid

NOTE

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

A thread can obtain the invocation handle of the procedure context preceding any other procedure context in the active call chain by calling a system library function defined as follows:

LIB_GET_PRIOR_INVO_HANDLE(INVOCATION_HANDLE)

Arguments:

INVOCATION_HANDLE	A procedure invocation handle

Function Value:

HANDLE	An invocation handle: A valid invocation handle of the previous invocation in the call chain. INVO_HANDLE_NO_MORE indicates that there is no prior procedure context. INVO_HANDLE_NULL indicates that the input argument was invalid.

Walking the Call Chain

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

The navigation is done by:

1. Given an instance of program state (which contains a register set), construct an invocation context block.
2. Call *LIB_GET_INVO_HANDLE* to get a procedure invocation handle for that context.
3. Call *LIB_GET_PRIOR_INVO_HANDLE* to get the previous invocation handle
4. Repeat the previous set until the end of the chain has been reached (as signified by INVO_HANDLE_NO_MORE being returned).

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

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

4.4.4 Obtaining the Context of a Procedure

Given an invocation handle, a thread can obtain the current context of any procedure invocation in the call chain by calling a system library function defined as follows:

LIB_GET_INVO_CONTEXT(INVOCATION_HANDLE, CONTEXT)

Arguments:

INVOCATION_HANDLE	Handle for the desired invocation.
CONTEXT	The address of an invocation context block into which the procedure context of the frame specified by INVOCATION_HANDLE will be written.

Function Value:

STATUS	A status value: STATUS_SUCCESS indicates that the context of the requested frame information was returned in the block specified by CONTEXT. STATUS_FRAME_NOT_FOUND indicates that there is no frame with the specified invocation handle. STATUS_FAILURE indicates that the arguments were otherwise invalid.

4.5 Transfer of control

This standard states that a standard call may be done in any way that presents the called routine with the required environment (see Section 1.4, Definitions). Although this is true it is believed that the vast majority of standard conforming external calls will be done with a common sequence of instructions and conventions in both the ALPHA-64 and the ALPHA-32 flavors of the ALPHA environment. This common set of call conventions is so pervasive that it is included for reference as part of this standard.

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

4.5.1 Call Conventions

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

- **Procedure Value**

  The calling procedure must pass to the called procedure its procedure value. This may be a statically or dynamically bound procedure value. To do this, R27 is loaded with the procedure value before control is transferred to the called procedure.

- **Return Address**

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

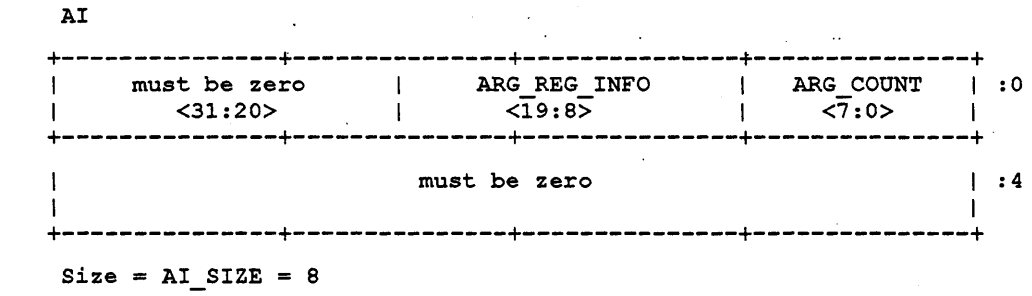
- **Argument List**

  The *argument list* is an ordered set of zero or more *argument items*, that together comprise a logically contiguous structure known as an *argument item sequence*. This logically contiguous sequence is in practice 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 R16..R21 and/or registers F16..F21. (See Section 5.2.2, Argument List Structure, for details of argument-to-register correspondence). The remaining items are collected in a memory argument list that is a naturally-aligned array of quadwords. In a standard call, this list (if present) must be passed at 0(SP).

- **Argument Information**

  The calling procedure must pass to the called procedure information about the argument list. This information is passed in the Argument Information Register (R25) which has the following format:

Figure 4–11: Argument Information Register Format



AI.ARG_COUNT<7:0> is an unsigned byte that represents the number of 64-bit argument items in the argument list (commonly referred to as the "argument count").

AI.ARG_REG_INFO<19:8> is an 12-bit bitvector consisting of 6 groups of 2 bits corresponding to the 6 arguments that may be passed in registers. These groups describe how each of the first six arguments are passed in registers with the first group (bits <9:8>) describing the first argument. Each group is encoded as follows:

Table 4–5: Argument Register Usage Encodings

Name	Value	Meaning
AR_I64	0	64-bit or 32-bit sign extended to 64-bit argument passed in an integer register *or* - Argument is not present *or* - Argument slot used by previous argument
AR_FV	1	VAX floating argument passed in a floating register
AR_FS	2	IEEE S floating argument passed in a floating register
AR_FT	3	IEEE T floating argument passed in a floating register

RATIONALE:

The above presents one solution to the "printf" problem mentioned below.

\\
There exists a certain type of procedure, such as the ULTRIX printf() function or the VMS FAO interfaces, which allow any argument list position to be of a "generic" type depending on the value of preceding arguments in the list. In particular, this type of procedure allows any argument list item to be of type integer or floating point passed by value and also might require the argument list to be addressable as a contiguous sequence of argument items.

This poses a problem in that, for any argument list item, the location of an in-register argument (integer or floating point register) can't be determined until the function call at run-time.

This problem does not exist on hardware systems where integer and floating point operations are done on the same set of registers (such as a conventional VAX).

This standard provides a mechanism to solve this problem by requiring that the caller pass information about the location of argument item in the argument information register.

This problem is solved on MIPS systems by the following technique: when any argument occurs that is known by definition to be non-floating, then all subsequent arguments, whether floating point passed by value or not, are passed in the integer argument locations.

This technique works because the argument(s) that determines the types of following arguments is never floating point, so such "generic" argument lists are always forced to the integer argument locations. This technique works reasonably well on architectures like MIPS, because there is a fast non-memory data path from the floating point coprocessor registers to the integer processor registers.

Since it's not possible on ALPHA to move data from the floating point registers to the integer registers except through memory, a severe performance penalty could be imposed by adopting techniques like that used for MIPS.
\\

Other bits<63:20> are reserved bits and must be zero.

- **Function Result**

  If a standard-conforming procedure is a function, and the function result is returned in a register, then the result is returned in R0, F0, or F0..F1. Otherwise, the function result is returned via the first argument item or dynamically as defined in Section 5.2.7, Returning Data.

- **Stack usage**

  SP must at all times denote an address that has at least the minimum alignment required by the ALPHA hardware. In addition, whenever control is transferred to another procedure the stack must be octaword aligned. (This has the side effect that the in memory portion of the argument list, if any, will start on an octaword boundary.) During a procedure invocation the SP may never be set to a value that is higher than the value of the SP at entry to that procedure invocation.

  RATIONALE:

  Because this standard requires that the SP be octaword aligned at the time of a call, compilers may arrange to have local (stack) data octaword aligned without costly extra code.

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

The stack pointer (SP) is used by the hardware in raising exceptions and asynchronous interrupts. This will cause the contents of the next 2048 bytes below the current SP value to be continually and unpredictably modified. No software that conforms to this standard may depend on the contents of the 2048 stack locations below 0(SP).

4.5.2 Linkage

A hardware architecture that has the property that instructions cannot contain full virtual addresses is sometimes 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.

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

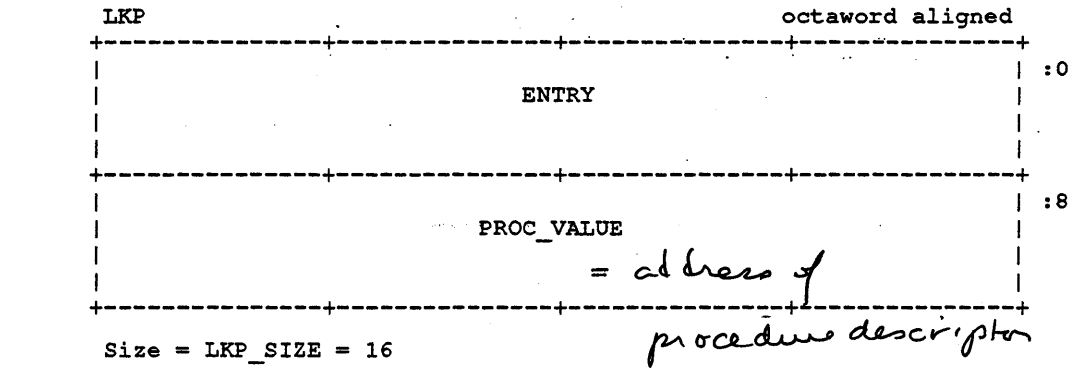
A procedure's linkage section includes the procedure descriptor for the procedure, addresses of all external variables and procedures referenced by the procedure, and other constants that a compiler may choose to reference using a linkage pointer.

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

Compilers usually arrange to have the environment setup code for bound procedures also load R27 with the address of the procedure's descriptor as part of the environment setup so that it may be used as a linkage pointer as described above. (See Section 4.5.4, Bound Procedure Values, for an example.)

Although not required, linkages to external procedures are typically represented in the calling procedure's linkage section as a *linkage pair*. A linkage pair, which should be octaword aligned, has the following format:

Figure 4–12: Linkage Pair Format



LKP.ENTRY is the absolute address of the first instruction of the called procedure's entry code sequence.

LKP.PROC_VALUE is the procedure value of the procedure to be called. This is normally the absolute address of a procedure descriptor for the procedure to be called but may be a bound procedure value in certain cases (such as for procedures that are called through certain types of transfer vectors).

```
\\
Because a code entry address may always be obtained by loading the address
pointed to by a procedure value + 8, only the procedure value is technically
required in the linkage section. The code entry address is also placed in the
linkage section to provide better performance for external procedure calls.
\\
```

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

As an example of typical code to call an external procedure Q as represented by a linkage pair, suppose that R4 is the register currently containing the address of the current procedure's descriptor. Q can be called by:

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

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

```
\\
The linkage section is part of the called procedure, and the layout of a
linkage section is determined by the compiler. Procedures compiled together
(belonging to a single object module) will share a linkage section that will
contain a procedure descriptor for each procedure in the module. Offsets for
procedure descriptors and address constants within the linkage section are
assigned by the compiler when the module is compiled.

In practice, the linkage section will be allocated in a program section that is
read-only from the point of view of the program. However when the image section
is loaded fixups will be done which will cause the image section to become
non-shared between processes. This read-only linkage section could also contain
all read-only constants defined by the module. However, all the read-only
constants in that section would then also become non-shared between processes.
If there are more than a page or two of read-only constants, this might
significantly increase physical memory usage; thus, caution must be exercised.
\\
```

4.5.3 Calling Computed Addresses

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

Suppose R4 contains such a computed procedure value (simple or bound). An example of the code to call the procedure that it describes is:

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

Compiled code may optionally be constructed to support enhanced compatibility by generating and using procedure signature information as described in Section 4.2, Procedure Signatures. In this mode calls to a computed address must use a slightly different calling sequence similar to the one described below:

```
        LDL     R28,0(R4)           ;Load the flags field of the target PDSC
        MOVQ    #AI_LITERAL,R25     ;Load Argument Information register
        SRL     R28,#PDSC_FLAGS_NO_JACKET,R28 ;Use jacketing ?
        BLBC    R28,CALL_JACKET     ;Yes
        LDQ     R26,8(R4)           ;Entry address to scratch register
        MOV     R4,R27              ;Procedure value to R27
        JSR     R26,(R26)           ;Call entry address.
        ...                         ;Rest of procedure code goes here

CALL_JACKET:                        ;Generated out of line
        LDQ     R26,JACKET(LP)      ;Entry address to scratch register
        LDQ     R27,JACKET+8(LP)    ;Load procedure value
        MOV     R4,R23              ;Pass address of routine to call
        LDQ     R24,CALL_OUT_INFO(LP) ;Pass address of call out information
        JSR     R26,(R26)           ;Call Jacket routine
        BR      back_in_line        ;Return to normal code path

\\
For the ALPHA-32 flavor of the calling standard the name of the jacket routine
is [TBS]. It receives two non-standard parameters in R23 and R24. For the
ALPHA-64 flavor of the calling standard use of this mode is not currently
defined.
\\
```

4.5.4 Bound Procedure Values

There are two distinct classes of procedure values; those that represent simple procedures and normally acquire their value sometime prior to execution of code (normally as part of a compile, link or activation process - see Section 3.2, Procedure Representation), and those that represent bound procedures and normally acquire their value at run time.

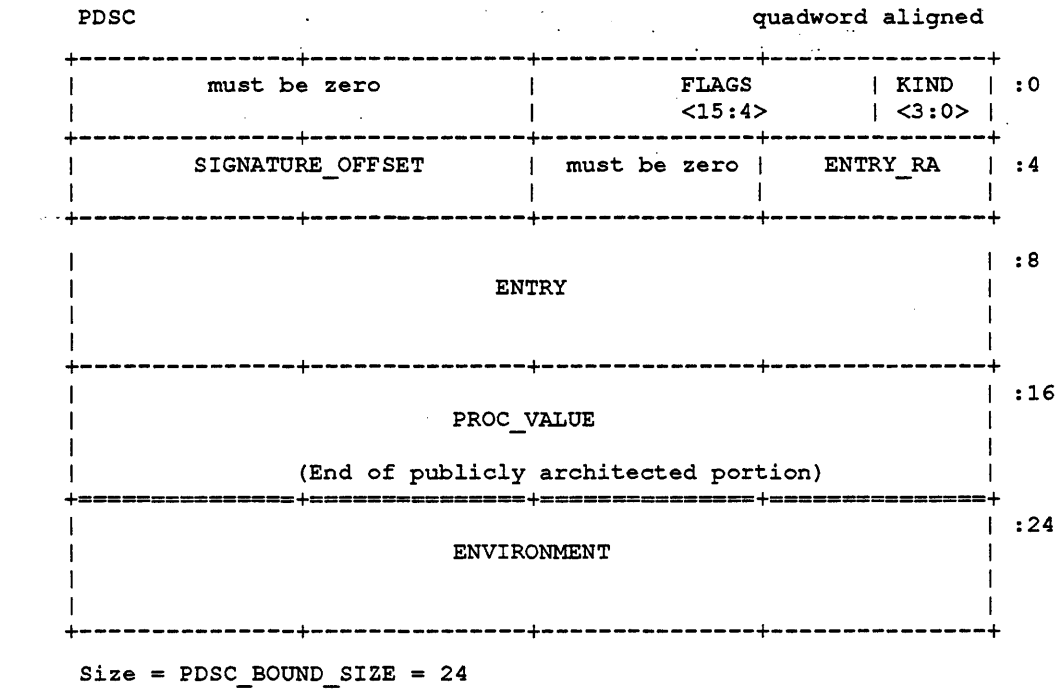
Bound procedure values, as defined by this standard, are designed for multilanguage use and utilize the properties of procedure descriptors to allow callers of procedures to use common code to call both bound and simple procedures.

The procedure value for a bound procedure is a pointer to a bound procedure descriptor which, like all other procedure descriptors, contains the address to which the calling procedure must transfer control at offset 8. This offset is required to allow the use of common code as mentioned above. Use of this common code will cause the pointer to the bound procedure descriptor to be passed in R27 in the same way that the simple procedure descriptor address would be passed for a simple procedure value (see Section 4.5.2, Linkage). It is then the responsibility of the setup code for the bound procedure to use this pointer to insure that whatever environment is required for the correct execution of the bound procedure is loaded before control is transferred to the bound procedure. (This usually involves loading R27 with a pointer a statically defined procedure descriptor for that procedure so that it may be used as a linkage pointer).

Given a procedure Q, if Q is not nested within another procedure, then a procedure value for Q is simply the address of Q's procedure descriptor. Careful inspection of the sample procedure value calling sequence given in Section 4.5.3, Calling Computed Addresses, will show that the code sequence simply calls Q.

If, however, Q is a subprocedure of a procedure P (i.e. a bound procedure), then a procedure value for Q must be a bound procedure value. The bound procedure value points to a special type of procedure descriptor that contains the information for both the transfer of control and accessing the up-level data. The format of this data structure is defined as follows:

Figure 4–13: Bound Procedure Descriptor Format



PDSC.KIND = 0 (denoted by PDSC_KIND_BOUND) specifies a bound procedure descriptor.

\\
For the FP based environment of the ALPHA-32 calling standard there are two values (0 and 8) of PDSC.KIND that must **never** be pointed to by a FP. This is because the algorithm for determining if the contents of FP points to a procedure descriptor directly or indirectly through a stack location is based on determining if the quadword which is pointed to has the low order 3 bits clear. Thus PDSC.KIND values of 0 or 8 fulfill this criteria when it would not be valid.

Bound procedure descriptors and null procedure descriptors are formatted identically for both the ALPHA-32 and the ALPHA-64 flavor of the calling standard. Since neither of these types of procedure descriptors can ever be pointed to by an FP they are assigned these two special values of PDSC.KIND, thus solving the problem.
\\

PDSC.FLAGS is a vector of flag bits which must be an exact copy of the flag bits contained in the quadword pointed to by PDSC.PROC_VALUE.

RATIONALE:

The above field is required so that if a calling routine needs the flag information it does not have to do the indirection to look at the flag bits of the actual bound procedure.

PDSC.ENTRY_RA is the number of the register in which the return address is passed to the environment setup code for this procedure.

Transfer code for bound procedures must follow the same rules as procedure entry code. (See Section 4.5.5, Entry and Exit Code Sequences, for details) In particular, since the environment setup code never establishes itself as the *current* procedure, the return address must remain in this register during the entire time the setup code is active. This implies that PDSC.ENTRY_RA in a bound procedure descriptor must be the same as the PDSC.ENTRY_RA of the procedure descriptor for the procedure for which the environment is established.

PDSC.SIGNATURE_OFFSET must be set to zero in all bound procedure descriptors.

RATIONALE:

Since bound procedure descriptors are created at run time there is a strong desire to minimize the overhead of doing that creation. In this case a routine which needs the signature information will first determine that the information is needed. On attempting to access the signature information the routine will discover that this offset is zero. It can then check the kind field and verify that it is indeed looking at a bound procedure descriptor at which point it can utilize the PDSC.PROC_VALUE field to find the procedure descriptor for which the bound procedure descriptor was created and then repeat the steps needed to find the signature information.

PDSC.ENTRY is the address of transfer code sequence

RATIONALE:

The offset of this address is tailored to match with the same offset in all procedure descriptors.

PDSC.PROC_VALUE is the procedure value of the procedure to be called by the transfer code. This can be either the address of a procedure descriptor for the procedure or possibly another bound procedure value.

PDSC.ENVIRONMENT is the environment value to pass to the procedure.

When the transfer code sequence addressed by PDSC.ENTRY is called (by a call sequence such as the one given in Section 4.5.3, Calling Computed Addresses), the procedure value will be in R27, and the transfer code must finish setting up the environment for the target procedure. The preferred location for this transfer code is directly preceding the bound procedure. This saves a memory fetch and a branching instruction as well as optimizing instruction caches and paging. (Also, see note below.)

An example of a such a transfer code sequence, for a target procedure that expects the environment value to be passed in R22 and a linkage pointer in R27, is:

```
        LDQ     R22,24(R27)     ;Environment value to R22
        LDQ     R27,16(R27)     ;Procedure descriptor address to R27
Q_ENTRY::                       ;Normal procedure entry code starts here
\\
If the transfer sequence does not directly precede the procedure entry
point then the following instructions, or equivalent must be done AFTER the
above instructions:

        LDQ     R28,8(R27)      ;Entry address to scratch register
        JMP     R31,(R28)       ;Jump to entry address.
\\
```

After the transfer code has been executed and control is transferred to Q's entry address, R27 contains the address of Q's procedure descriptor, R26 (unmodified by transfer code) contains the return address, and R22 contains the environment value.

The environment value is needed by Q's code for such things as references to variables in P, and nonlocal GOTOs to points in P. When a bound procedure value such as this is needed, the bound procedure descriptor will normally be allocated on the parent's stack.

```
\\
Bound procedure descriptors must usually be generated at run-time since the
environment value is not usually known until the parent has been invoked.
\\
```

NOTE

To facilitate calls through bound procedure values or constructs that behave in a similar fashion (such as certain types of transfer vectors) this standard *requires* that a reserved instruction be placed at the address that is exactly 4 bytes before any external entry point to a procedure even if it is not a bound procedure. This reserved instruction is:

```
LDQ     R27,16(R27)     ;Procedure descriptor address to R27
```

and serves to setup R27 in those cases where it was otherwise used during the transfer process.

4.5.5 Entry and Exit Code Sequences

This section describes 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 which produces the defined effects is legal.

Entry Code Sequence

Since the value of the PC defines the current procedure, all properties of the environment specified by a procedure's descriptor must be valid before the first instruction beyond the procedure prolog (as defined by PDSC.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. In particular this means that 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 call must contain their original (at entry) contents.
- The register designated by PDSC.ENTRY_RA (R26 in a standard call) must contain its original (at entry) contents. *Note: This must be true even for non standard procedures to allow proper unwinding*
- No standard calls may be made

NOTE

If an exception is raised or an exception occurs in the prolog of a procedure, that procedure's exception handler (if any) will not be invoked since the procedure is not *current* yet. This implies that if a procedure has an exception handler compilers

may not move code into the procedure prolog that might cause an exception that would be handled by that handler.

When a procedure is called, the code at the entry address must synchronize (as needed) any pending exceptions caused by instructions issued by the caller, save the caller's context, and make the called procedure current (by executing the first instruction beyond the procedure prolog).

This is done by performing the following actions:

Perform step 1, then steps 2 and 3 in any order, then steps 4 - 6 in any order.

1. If PDSC.SIZE is not 0, then set register SP = SP – PDSC.SIZE.

   After any necessary calculations and stack limit checks, this step must be completed by exactly one instruction that modifies SP, which must be the instruction specified by PDSC.SP_SET.

2. If PDSC.KIND = PDSC_KIND_PC_STACK store the registers specified by PDSC.IREG_MASK and PDSC.FREG_MASK in the register save area as denoted by PDSC.RSA_OFFSET.

3. If PDSC.KIND = PDSC_KIND_PC_STACK, copy the return address to the quadword at PDSC.RSA_OFFSET.SAVED_RETURN.

4. If PDSC.FLAGS.BASE_REG_IS_FP is 1, copy register SP to register FP.

5. If PDSC.KIND = PDSC_KIND_PC_REGISTER, copy the return address to the register specified by PDSC.SAVE_RA if it is not already there.

6. Execute DRAINT if required (see Section 6.1.11, Other Considerations, for details)

Frame Pointer Conventions

After procedure prologue completion, the register indicated by PDSC.FLAGS.BASE_REG_IS_FP must contain the *frame pointer* of the stack frame, which 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.SIZE subtracted from the value of the stack pointer upon procedure entry.

For fixed frame procedures, the frame pointer is the stack pointer, which is not modified by that procedure after the instruction in that procedure prologue specified by PDSC.SP_SET.

Entry Code Example for a Stack Frame

This example assumes that this is a stack frame procedure, that registers R2..R4 and F2..F3 are saved and restored, that PDSC.RSA_OFFSET = 16, that the procedure has a static exception handler that does not reraise arithmetic traps, and that the procedure uses a fixed amount of stack (PDSC.FLAGS.BASE_REG_IS_FP is 0).

```
LDA     SP,-SIZE(SP)     ;Allocate space for new stack frame.
STQ     R26,16(SP)       ;Save return address
STQ     R2,24(SP)        ;Save first integer register.
STQ     R3,32(SP)        ;Save next integer register
STQ     R4,40(SP)        ;Save next integer register
STQ     FP,48(SP)        ;Save R29
STT     F2,56(SP)        ;Save first floating point register
STT     F3,64(SP)        ;Save last floating point register
DRAINT                   ;Force any pending hardware exceptions to be
                         ; raised. (see Section 6.1.11)
;Called procedure is now the current procedure.
```

Entry Code Example for a Register Frame

The following entry code example is for a called procedure that has no static exception handler, both PDSC.SAVE_RA and PDSC.ENTRY_RA specify R26 and the procedure utilizes a fixed amount of stack storage (PDSC.FLAGS.BASE_REG_IS_FP is 0).

```
LDA     SP,-SIZE(SP)     ;Allocate space for new stack frame.
;Called procedure is now the current procedure.
```

Exit Code Sequence

The end of procedure entry code can be determined easily given a PC value together with the PDSC.ENTRY_LENGTH value. Since there may be multiple return points from a procedure, detecting that a procedure exit sequence is being executed is not quite as straight forward. Unwind support routines *must* be able to detect if the stack pointer has been reset or not and if not, how to reset it. This is done by using a reserved instruction sequence.

Reserved Instruction Sequence for Procedure Exit

To allow the stack to be properly restored during an unwind the following reserved instruction **must** appear at every exit point from any procedure that uses stack (PDSC.SIZE ≠ 0):

```
        RET     R31,(Rn),signature_hint ;Return to caller with signature

\\
We intend to propose an ECO to the ALPHA SRM to reserve bits <13:10> of
the hint field when the RET hint (10) is used. These bits set to 0001 will
constitute the above mentioned "signature hint".
\\
```

Furthermore, for any such procedure that does not return a value on the stack (PDSC.FUNC_RETURN ≠ FR_STK) the above instruction must be *directly* preceded by one of the two reserved stack resetting instruction as in:

```
LDA     SP,SIZE(SP)      ;Reset stack
RET     R31,(Rn),signature_hint ;Return to caller with signature
```

or

```
ADDQ    Rn,SP,SP         ;Reset stack
RET     R31,(Rn),signature_hint ;Return to caller with signature
```

Furthermore, for any such procedure that has PDSC.FLAGS.BASE_REG_IS_FP set to 1, the resulting sequence must be directly preceded by the FP reloading instruction as in:

```
LDQ     FP,FP_OFFSET(SP) ;Restore FP if needed
LDA     SP,SIZE(SP)      ;Or ADDQ Rn,SP,SP to Reset stack
RET     R31,(Rn),signature_hint ;Return to caller with signature
```

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

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

- If the PC points at a RET R31,(Rn) instruction that has the calling standard reserved signature bits set in the hint field of the instruction then SP has already been reset, registers have already been restored and the unwind can proceed.
- If the PC points to either a LDA SP,nnn(SP) or an ADDQ Rn,SP,SP instruction that is immediately followed by the instruction described previously and nnn is the same value as is stored in PDSC.SIZE, then registers have already been restored but the SP must be incremented by PDSC.SIZE before the unwind can proceed.
- If the PC points to a LDQ FP,xxx(SP) instruction that is immediately followed by the instructions described previously and PDSC.KIND = PDSC_KIND_PC_STACK, then 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, SP must be incremented by PDSC.SIZE and the unwind can proceed.
- Otherwise, the registers must be restored, SP reset, and the unwind can proceed.

To allow this mechanism to operate correctly procedures that conform to this standard may not use either of the above mentioned instruction sequences in any case other that as described above.

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.

This is done by performing the following actions:

Perform steps 1 - 4 in any order, followed by steps 5 - 7 in that order.

1. If PDSC.FLAGS.BASE_REG_IS_FP is 1 and PDSC.FUNC_RETURN ≠ FR_STK, then copy FP to SP
2. If PDSC.KIND = PDSC_KIND_PC_STACK, and any registers were saved, then reload those registers from the register save area as specified by PDSC.RSA_OFFSET *except* FP if PDSC.FLAGS.BASE_REG_IS_FP is 1. If PDSC.FLAGS.BASE_REG_IS_FP is 0 and FP was saved then it should be reloaded in this step.
3. If PDSC.KIND = PDSC_KIND_PC_STACK, load a scratch register with the return address from the register save area as specified by PDSC.RSA_OFFSET. (If PDSC.KIND = PDSC_KIND_PC_REGISTER, the return address is already in scratch register PDSC.SAVE RA.)
4. Execute DRAINT if required (see Section 6.1.11, Other Considerations, for details)
5. If PDSC.KIND = PDSC_KIND_PC_STACK and PDSC.FLAGS.BASE_REG_IS_FP is 1, reload R29 (FP) from the saved FP in the register save area.

   After any necessary calculations, this step must be completed by exactly one instruction as described above.
6. If a function value is not being returned at 0(SP) (PDSC.FUNC_RETURN ≠ FR_STK), then restore SP to the value it had at procedure entry by adding the value that was

stored in PDSC.SIZE to SP. (In some cases the returning procedure will leave SP pointing to a lower stack address than it had on entry to the procedure, as specified in Section 5.2.7, Returning Data).

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

7. Execute the instruction RET R31,(Rn) instruction as described above to return control to the calling procedure.

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 may choose to defer their removal due to optimizations or other considerations.

Exit Code Example for a Stack Frame

The following is the return code sequence for the stack frame example above:

```
LDQ     R28,16(SP)        ;Get return address
LDQ     R2,24(SP)         ;Restore first integer register.
LDQ     R3,32(SP)         ;Restore next integer register
LDQ     R4,40(SP)         ;Restore next integer register
LDT     F2,56(SP)         ;Restore first floating point register
LDT     F3,64(SP)         ;Restore last floating point register
DRAINT                    ;Force any pending hardware exceptions to be
                          ; raised.  (see Section 6.1.11)
LDA     SP,SIZE(SP)       ;Restore SP (size is compiled into PDSC.SIZE)
RET     R31,(R28),signature_hint ;Return to caller with signature
```

Exit Code Example for a Register Frame

The following is the return code sequence for the register frame example above:

```
LDA     SP,SIZE(SP)       ;Restore the SP
RET     R31,(R26),signature_hint ;Return to caller with signature
```

CHAPTER 5

DATA MANIPULATION

This section deals with the storage and passing of data.

5.1 Data Types

```
\\
The ALPHA-64 does not used the notion of data type as a defined part
of the standard as it was for VAX/VMS. This is because the data type is no
longer stored in ALPHA-64 descriptors. Nor does the ALPHA SRM define hardware
data types in the same way as was done for the original VAX architecture.
However, the concept of data types seems to be a useful one for purposes of
explanation, and as a means of direct comparison with conventional VAX. For
this reason the concept is presented and used in this document.
\\
```

5.1.1 Argument and Function Value Data Types

Each data type implemented for a higher-level language may be categorized into one of the following classes of data types for procedure parameters and elements of file records:

- Atomic
- String
- Miscellaneous

When existing data types are not sufficient to satisfy the semantics of a language, new data types are added to this standard, including certain language-specific ones.

Each data type code presented in the sections below indicates a unique data format. These encodings should be used whenever there is a need to identify data types to achieve as much commonality as possible amongst user software.

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

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

Fully Supported Data Types

The following data types are fully supported for use as standard argument data types, and may be passed between external procedures and between languages.

These data types may be freely utilized in the ALPHA-64 environment.

```
\\
Each of the data types represented below is represented on VAX/VMS by
a symbol of the form DSC$K_DTYPE_<Type Designator>.
\\
```

Atomic Data Types

Table 5–1 shows how atomic data types are defined and encoded.

Table 5–1: Atomic Data Types

Type Designator	Code	Natural Alignment	Name/Description
Z	0	quadword	Unspecified The calling program has specified no data type. The default argument for the called procedure should be the correct type.
BU	2	byte	Byte logical (unsigned) 8-bit unsigned quantity.
WU	3	word	Word logical (unsigned) 16-bit unsigned quantity.
LU	4	longword	Longword logical (unsigned) 32-bit unsigned quantity.
QU	5	quadword	Quadword logical (unsigned) 64-bit unsigned quantity.
OU[1]	25	octaword	Octaword logical (unsigned) 128-bit unsigned quantity.
B	6	byte	Byte integer (signed) 8-bit signed 2's-complement integer.
W	7	word	Word integer (signed) 16-bit signed 2's-complement integer.
L	8	longword	Longword integer (signed) 32-bit signed 2's-complement integer.
Q	9	quadword	Quadword integer (signed) 64-bit signed 2's-complement integer.
O[1]	26	octaword	Octaword integer (signed) 128-bit signed 2's-complement integer.
F	10	longword	F_floating 32-bit F_floating quantity representing a single-precision number. See ALPHA SRM for detail of representation.

[1]These data types may be utilized but it should be noted that the conventional VAX hardware architecture provides special support for these data types which the ALPHA hardware does not provide. Therefore, language or application utilization of these data types may result in (sometimes severe) performance penalties caused by software handling of these types.

Table 5–1 (Cont.): Atomic Data Types

Type Designator	Code	Natural Alignment	Name/Description
D	11	quadword	D_floating 64-bit D_floating quantity representing a double-precision number. See ALPHA SRM for detail of representation.
G	27	quadword	G_floating 64-bit G_floating quantity representing a double-precision number. See ALPHA SRM for detail of representation.
FC	12	longword	F_floating complex Ordered pair of F_floating quantities, representing a single-precision complex number. The lower addressed quantity is the real part, the higher addressed quantity is the imaginary part.
DC	13	quadword	D_floating complex Ordered pair of D_floating quantities, representing a double-precision complex number. The lower addressed quantity is the real part, the higher addressed quantity is the imaginary part.
GC	29	quadword	G_floating complex Ordered pair of G_floating quantities, representing a double-precision complex number. The lower addressed quantity is the real part, the higher addressed quantity is the imaginary part.
FS	[TBS]	longword	IEEE S_floating 32-bit IEEE S_floating quantity representing a single-precision number. See ALPHA SRM for detail of representation.
FT	[TBS]	quadword	IEEE T_floating 64-bit IEEE T_floating quantity representing a double-precision number. See ALPHA SRM for detail of representation.
FSC	[TBS]	longword	IEEE S_floating complex Ordered pair of IEEE S_floating quantities, representing a double-precision complex number. The lower addressed quantity is the real part, the higher addressed quantity is the imaginary part.
FTC	[TBS]	quadword	IEEE T_floating complex Ordered pair of IEEE T_floating quantities, representing a double-precision complex number. The lower addressed quantity is the real part, the higher addressed quantity is the imaginary part.

String Data Types

String data types are ordinarily described by a string descriptor. Table 5–2 shows how the string data types are defined and encoded.

Table 5–2: String Data Types

Type Designator	Code	Natural alignment	Name/Description
T	14	byte	Character string A single 8-bit character (atomic data type) or a sequence of 8-bit characters (string data type).
VT	37	word	Varying character string A 16-bit unsigned count of the current number of 8-bit characters following, followed by a sequence of 0 to 2^{16}-1 8-bit characters.
NU	15	byte	Numeric string, unsigned
NL	16	byte	Numeric string, left separate sign
NLO	17	byte	Numeric string, left overpunched sign
NR	18	byte	Numeric string, right separate sign
NRO	19	byte	Numeric string, right overpunched sign
NZ	20	byte	Numeric string, zoned sign
P[1]	21	byte	Packed-decimal string
V[1]	1	byte	Aligned bit string A string of contiguous bits. The first bit is bit <0> of the first byte, and the last bit is any bit in the last byte. Remaining bits in the last byte must be zero on read and are cleared on write.
VU[1]	34	N/A	Unaligned bit string The data is a sequence of contiguous bits located arbitrarily with respect to byte boundaries. See also aligned bit string (V) data type.

[1]These data types may be utilized but it should be noted that the conventional VAX hardware architecture provides special support for these data types which the ALPHA hardware does not provide. Therefore, language or application utilization of these data types may result in (sometimes severe) performance penalties caused by software handling of these types.

Miscellaneous Data Types

Table 5–3 shows how miscellaneous data types are defined and encoded.

Table 5–3: Miscellaneous Data Types

Type Designator	Code	Natural Alignment	Name/Description
None.			

Unsupported Data Types

The following tables is included only for information. It represents VAX/VMS data types that are not supported in this environment.

Table 5–4: Unsupported Data Types

Type Designator	Code	Name/Description
ZI	22	Sequence of VAX instructions
DSC	24	Descriptor
H[1]	28	H_floating
HC[1]	30	H_floating complex
CIT	31	COBOL Intermediate Temporary
BPV	32	VAX/VMS Bound procedure value
BLV	33	Bound label value
ADT	35	Absolute date and time

[1]H_floating data is not supported for general usage in either flavor of the ALPHA calling standard. However, conversion routines will be supplied to allow users to convert existing H_floating data to other storage formats.

Varying Character String Data Type

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

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

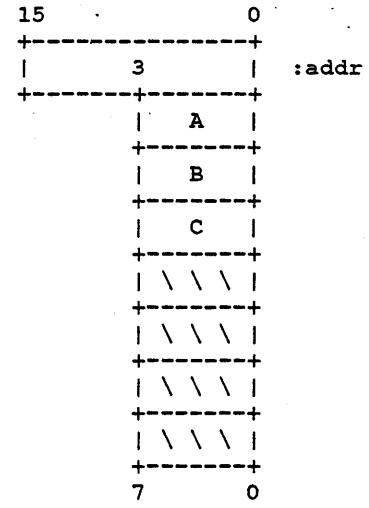
BODY — A fixed-length area containing the string that can vary from zero to a maximum length defined for each instance of string. The range of this maximum length is 0 to 2^{16}-1.

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

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

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

Figure 5–1: VT Data Format



5.1.2 Argument Descriptors

Goals

The goals for the design of ALPHA-64 descriptors include the following:

- Support high level languages

  Support passing arguments between procedures that conform to the ALPHA-64 calling standard. Thus, they are designed to support high level language (HLL) procedures calling procedures that are:

  - written in a HLL (same language or a different language)
  - part of the Run Time Library
  - part of a callable utility

- Language Level VAX/VMS compatibility

  VAX/VMS programs that use descriptors via mechanisms such as %DESCR will continue to work in the ALPHA-64 environment (*except* those programs that pass *atomic* data by %DESCR).

- Make descriptors self-defining

  Given the starting address, it is possible to determine what fields exist in a descriptor.

- Minimize time of access to data

  Descriptors frequently need to trade off size of the descriptor for speed of access to descriptor data. ALPHA-64 descriptors favor improving the speed of access to data.

- Avoid redundant information

  Several VAX/VMS descriptors contain fields that can be derived from other fields in the descriptor. ALPHA-64 descriptors are designed to avoid that redundant information.

- Avoid redundant forms

Due to evolution of the VAX/VMS calling standard, there are multiple ways of describing some objects with a descriptor. ALPHA-64 descriptors avoid redundant forms.

- Support 8-bit and multibyte character sets

  ALPHA-64 descriptors support 8-bit character sets and multibyte character sets.

- Avoid unsupported combinations of fields

  ALPHA-64 descriptors are designed to avoid unsupported combinations of fields. VAX/VMS descriptors have 2 fields, CLASS and DTYPE, that are not entirely orthogonal to one another. Combinations that seem quite feasible are not supported and often go undetected by DIGITAL software. A crisper definition of descriptors is applied to the ALPHA-64 environment.

- Provide uniform array descriptors

  On VAX/VMS, FORTRAN will create a contiguous array descriptor for an array of real values whereas PASCAL will create a noncontiguous array descriptor for the same array. This problem is resolved by avoiding special descriptor classes for contiguous arrays. Noncontiguous array descriptors can describe both contiguous arrays and cross sections of arrays that cannot be described by a contiguous array descriptor.

- Provide uniform bit descriptors

  Languages can check whether a bit string or array is aligned by testing the lowest three bits of the POSITION field in the descriptor.

- Provide uniform descriptors for sequences of bits and characters

  On VAX/VMS, some languages pass strings using string class descriptors, and expect string class descriptors for input arguments; other languages use one dimensional array descriptors. This problem is resolved by having one form of descriptor for linear, one-dimensional data.

- Avoid addressing calculations in called procedures

  The addressing fields in the ALPHA-64 array descriptors are optimized for compiled code referencing an array element.

Non-Goals

The following are non-goals for the ALPHA-64 descriptor design:

- Descriptors are not designed to describe objects to the degree needed by tools such as a debugger or a data dictionary.

  These tools should use their own data structures, such as DSTs, to describe objects. The calling standard will not encompass all the requirements of such utilities.

- VAX/VMS and ALPHA-64 descriptors are not identical.

  ALPHA-64 descriptors support high level language use of descriptors. Code that explicitly declares and references fields in VAX/VMS descriptors may not work in the ALPHA-64 environment.

Use of Descriptors for Procedure Arguments

Standard descriptors are defined for passing argument information between high level languages.

Unless explicitly stated otherwise in this standard, the calling procedure must fill in all fields in a descriptor. This is true whether the descriptor is generated by default or by a language extension. The fields must be filled in even if a called procedure written in the same language would ignore the contents of some of the fields.

Unless explicitly stated otherwise in this standard, all fields of an ALPHA-64 descriptor represent unsigned quantities. Descriptors are read-only from the point of view of the called procedure, and may be allocated in read-only storage at the option of the calling procedure (except by agreement between the calling and called procedure, such as dynamic text string descriptors). Memory containing the descriptor is owned by the calling procedure and may be used for other purposes after the call.

Descriptors must be quadword aligned.

Procedure argument descriptors are divided into two broad categories:

- descriptors for byte-addressable arguments
- descriptors for bit-addressable arguments

Descriptors for Byte-addressable Arguments

The broad class of descriptors for byte-addressable arguments contains the following specific descriptor kinds:

- Fixed text
- Varying text
- Dynamic text
- Byte-addressable array

ALPHA-64 byte-addressable array descriptors may be used to pass arrays where each array element starts and ends on a byte boundary. The storage of the array elements is allocated with a fixed (possibly zero) number of bytes separating logically adjacent elements.

The four kinds of byte-addressable argument descriptors support all the types of text strings understood in the ALPHA-64 software architecture. Each of the languages with a concept of text string must map it to one of these forms. Utility routines that accept text string arguments by descriptor should anticipate receiving text strings in any of these forms, and only these forms.

Figure 5–2: Byte-addressable Argument Descriptor Format

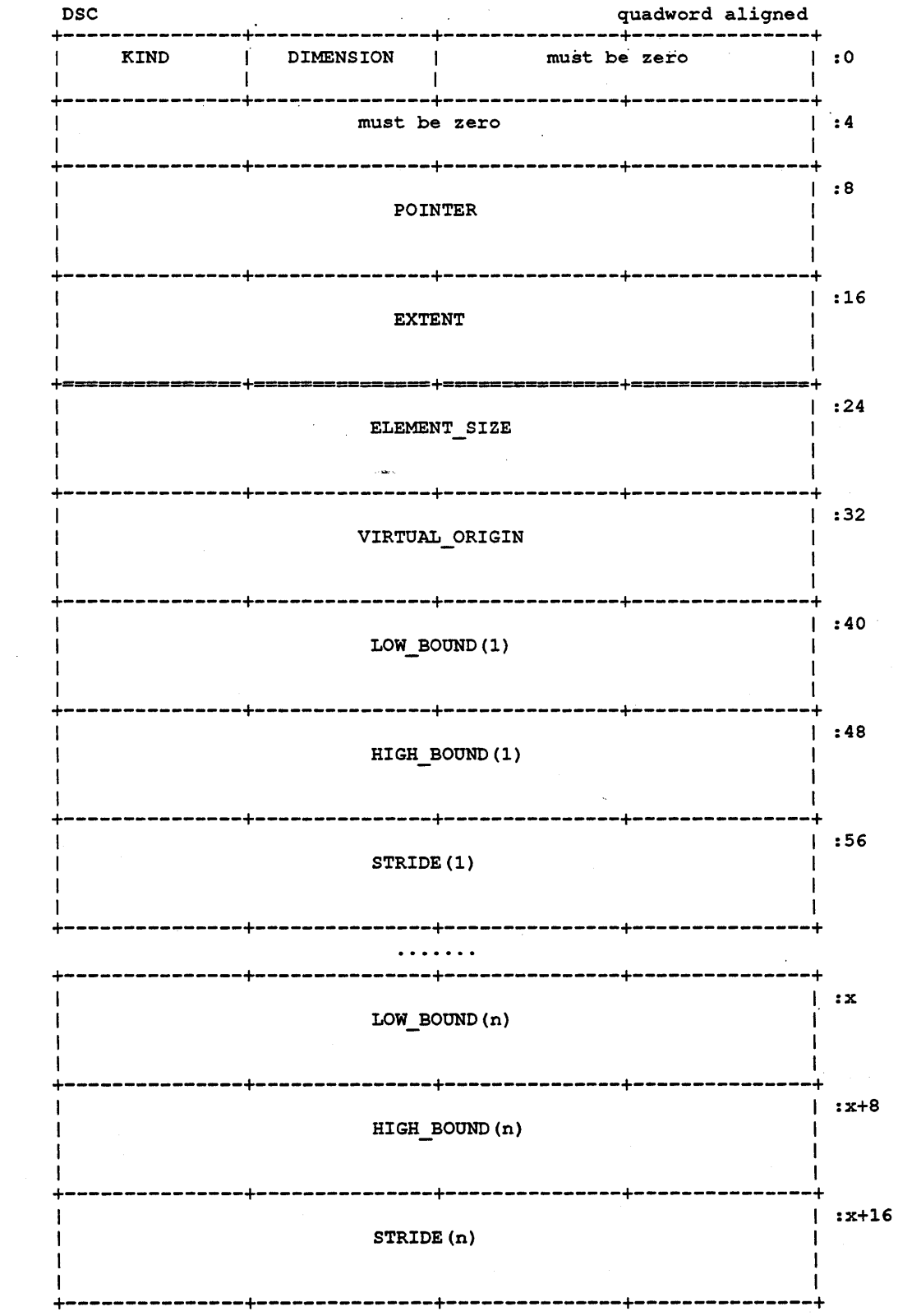


Figure 5–2 Cont'd on next page

Figure 5–2 (Cont.): Byte-addressable Argument Descriptor Format

```
Size = DSC_BYTE_SIZE = 24                    x = 24*(n - 1) + 40
```

DIMENSION is an unsigned integer 1..255 specifying the number of dimensions in an array. If DSC.KIND is not DSC_KIND_BYTE_ARRAY, then DSC.DIMENSION must be zero. (This field is used in the calculation of the size of the descriptor).

KIND is the descriptor kind attribute:

DSC_KIND_FIXED_TEXT = 65
DSC_KIND_DYNAMIC_TEXT = 66
DSC_KIND_VARYING_TEXT = 67
DSC_KIND_BYTE_ARRAY = 68

RATIONALE:

DSC.KIND overlays the VAX/VMS DSC$B_CLASS field in VAX/VMS descriptors. KINDs don't use same numbers as VAX/VMS descriptor classes so that they may be differentiated.

POINTER is the address of the first byte of storage occupied by the data. This is the address of the first byte of the string or, for an array, the address of element (DSC.LOW_BOUND(1), . . . DSC.LOW_BOUND(n)).

EXTENT specifies the number of bytes of contiguous storage occupied by the data.

When DSC.KIND = DSC_KIND_FIXED_TEXT or DSC.KIND = DSC_KIND_DYNAMIC_TEXT, then DSC.EXTENT is an integer $0..2^{64} - 1$ specifying the length in bytes of the string.

When DSC.KIND = DSC_KIND_VARYING_TEXT, then DSC.EXTENT is an integer $0..2^{16} - 1$ specifying the maximum length in bytes of the varying string.

When DSC.KIND = DSC_KIND_BYTE_ARRAY, and DSC.DIMENSIONS = 1, and DSC.STRIDE = DSC.ELEMENT_SIZE, then DSC.EXTENT is an integer $0..2^{64} - 1$ specifying the length in bytes of the storage occupied by the array.

For all other cases (multidimensional arrays), the contents of DSC.EXTENT are *unpredictable.*

ELEMENT_SIZE specifies the number of bytes of storage occupied by each element of a character array, and is present if and *only* if DSC.KIND = DSC_KIND_BYTE_ARRAY.

For arrays of fixed length text strings, DSC.ELEMENT_SIZE is an integer $0..2^{64} - 1$ specifying the length in bytes of each element of the array.

For arrays of varying text strings, DSC.ELEMENT_SIZE is an integer $0..2^{16} - 1$ specifying the maximum length in bytes of each element of the array.

For all other array types (multidimensional), the contents of DSC.ELEMENT_SIZE are *unpredictable.*

RATIONALE:

ELEMENT_SIZE is there because some languages require the argument passing capabilities provided by ELEMENT_SIZE: PL/I (*,*) CHARACTER(*) and (*,*) BIT(*) for example. Those argument types will be language-specific without ELEMENT_SIZE in the general descriptor design.

VIRTUAL_ORIGIN is the address of element (0, 0, . . . 0), and is present if and *only* if DSC.KIND = DSC_KIND_BYTE_ARRAY. This address is not necessarily within the storage of the array. In extreme cases it may not even be dereferenced as it may lie outside of the current address space.

RATIONALE:

VIRTUAL_ORIGIN in array descriptors could be derived from POINTER, but it seems right to provide it anyway.

Called procedures will usually need to have the virtual origin, since indexing will normally be done relative to virtual origin. If VIRTUAL_ORIGIN were not provided in addition to POINTER, then most called procedures would have to do the calculations, on every call, to get the virtual origin. If VIRTUAL_ORIGIN were not provided, the WORST case is that the caller computes it on every call, which is the same cost in run time as having the called routines compute it at every call; the more usual case is that the caller can calculate VIRTUAL_ORIGIN just once and use it for many calls, thus saving the cost of calculating the virtual origin for each call. The extra storage required for this seems worth it.

If, and *only* if, DSC.KIND = DSC_KIND_BYTE_ARRAY, then the following fields occur once per dimension:

LOW_BOUND(i) is a signed integer specifying the lower bound of the *i*th dimension of the array.

HIGH_BOUND(i) is a signed integer specifying the upper bound of the *i*th dimension of the array.

STRIDE(i) is a signed integer specifying the difference in bytes between the addresses of successive elements of the *i*th dimension of the array.

Data Type Conventions for Byte-addressable Argument Descriptors

Byte-addressable argument descriptors do not specify the data type contained by a string or array. This must be established by agreement between calling and the called procedures.

All characters in a fixed, varying, or dynamic string must be of the same size, and the character size cannot be determined by inspecting the descriptor; the calling and called procedures must agree on both the character size and the character set.

Run Time Conventions for Byte-addressable Argument Descriptors

The conventions controlling the use of byte-addressable argument descriptors are determined by the kind of the descriptor.

- DSC_KIND_FIXED_TEXT

  A fixed text string is sequence of $0..2^{64} - 1$ bytes with a fixed extent.

  DSC.EXTENT specifies the number of bytes (not characters) occupied by the string.

When a fixed text string is written, the contents of the descriptor must not be modified.

- DSC_KIND_VARYING_TEXT

  A varying text string is an unsigned 16-bit integer immediately followed by a fixed length area containing $0..2^{16} - 1$ bytes. The varying string is left justified within this fixed length area.

  DSC.EXTENT, which must be $\leq 2^{16} - 1$, specifies the length in bytes (not characters) of the fixed length area.

  DSC.POINTER specifies the address of the unsigned 16-bit integer count that precedes the fixed length area. This integer, that must be $\leq$ DSC.EXTENT, specifies the current length in bytes of the varying string.

  When a varying text string is written, the contents of the descriptor must not be modified. If a text string is copied to a varying text string of smaller extent, the string must be truncated. When a varying text string is modified, the new current length of the string is written to the 16-bit count.

- DSC_KIND_DYNAMIC_TEXT

  A dynamic text string is a sequence of $0..2^{64} - 1$ bytes.

  An uninitialized dynamic text descriptor, and a descriptor for a null dynamic string, has a DSC.EXTENT of zero.

  The DSC.EXTENT and DSC.POINTER fields of a dynamic text descriptor may be changed when the associated dynamic string is modified. These fields are volatile across external procedure calls, and descriptors for dynamic text strings must be allocated in read/write memory.

  The system supplies functions for allocating and deallocating memory for dynamic strings. These functions may modify the DSC.EXTENT and DSC.POINTER fields of the descriptor. With one exception, these system supplied functions are the *only* procedures that may modify these fields.

  The only exception to this rule is that a procedure with knowledge that the descriptor has not yet been used must zero the extent field to mark it as being uninitialized. A procedure with this knowledge is typically the procedure that allocated the descriptor.

- DSC_KIND_BYTE_ARRAY

  A byte-addressable array is a ordered sequence of elements, where all elements of any array must be the same byte-addressable ALPHA-64 data type.

  All elements of an array must reside in distinct storage. That is, an ALPHA-64 array descriptor must not specify that any array elements overlap one another.

  When an array is written, the contents of the descriptor must not be modified.

  The address of element (i,j,k) of a byte-addressable array can be computed as follows.

  address = VIRTUAL_ORIGIN + i*STRIDE(1) + j*STRIDE(2) + k*STRIDE(3)

  The address of the first byte of a byte-addressable array is:

  address =

  VIRTUAL_ORIGIN +
  LOW_BOUND(1)*STRIDE(1) +
  LOW_BOUND(2)*STRIDE(2) +

LOW_BOUND(3)*STRIDE(3)

or

address = POINTER

RATIONALE:

There were two directions to go in unifying fixed text and one dimensional character array descriptors:

1. **Specify that DSC_KIND_FIXED_TEXT, DSC_KIND_VARYING_TEXT, and DSC_KIND_DYNAMIC_TEXT descriptors apply to all character sizes.**
2. **Specify that these descriptor kinds apply only to 8-bit characters, and that the standard would be extended for multibyte characters when necessary.**

The first solution implies that the size of character strings passed by descriptor cannot be determined by inspection at run time, and that interfaces that accept character strings must therefore be character-size-specific.

The second solution is more compatible with VAX. However, while sequences of characters passed by string descriptor could have their character sizes distinguished by inspection, the SAME sequence of characters passed to the SAME interfaces by one-dimensional array descriptor could not have their character size distinguished (because we have no data type field in descriptors). This would be an odd property.

Solution 1 seems to be the best one (especially since interfaces that accept character strings by reference must be character-size-specific in any case).

Descriptors for Bit-addressable Arguments

The broad class of descriptors for bit-addressable arguments contains the following specific descriptor kinds:

- Bit string
- Bit-addressable array

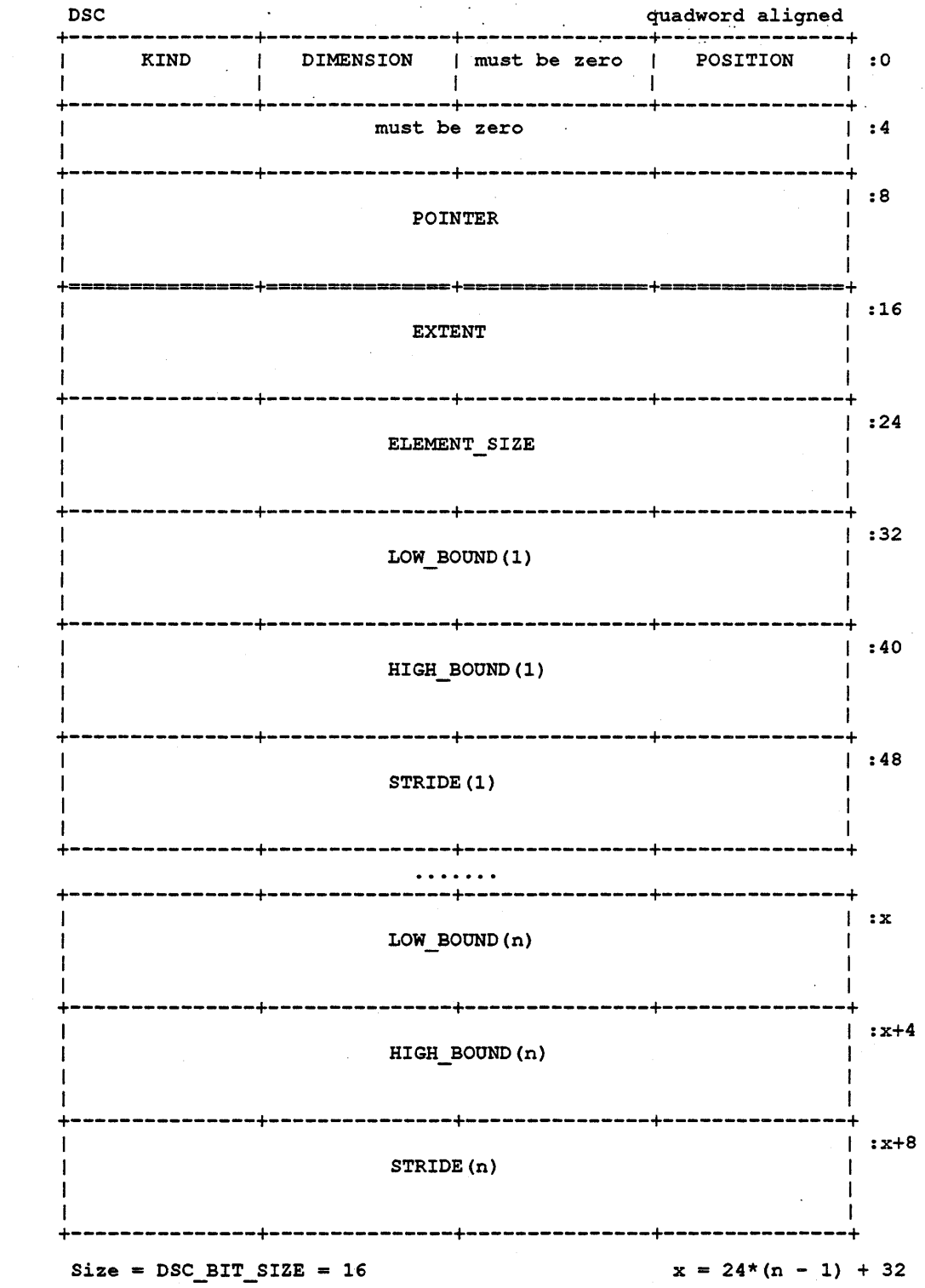
ALPHA-64 bit-addressable arguments descriptors may be used to pass bit sequences that start on an arbitrary bit boundary and end on an arbitrary bit boundary. Bit sequences that are byte aligned can be distinguished by position<2:0> = 000.

A bit-addressable arguments descriptor is capable of describing a bit sequence that starts anywhere in virtual memory. It is not capable of describing all of virtual memory as a single bit string or array.

ALPHA-64 bit-addressable array descriptors may be used to pass arrays where each array element may start on an arbitrary bit boundary and end on an arbitrary bit boundary. The storage of the array elements is allocated with a fixed (possibly zero) number of bits separating logically adjacent elements.

The two kinds of bit-addressable argument descriptors support all the types of bit strings understood in the ALPHA-64 software architecture. Each of the languages with a concept of bit string must map it to one of these forms. Utility routines that accept bit string arguments by descriptor should anticipate receiving bit strings in either of these forms, and only these forms.

Figure 5–3: Bit-addressable Argument Descriptor Format



POSITION is a signed integer 0..63 specifying the relative bit position with respect to DSC.POINTER of the first bit of the string or, for an array, the relative bit position with respect to DSC.POINTER of element (DSC.LOW_BOUND(1), . . . DSC.LOW_BOUND(n)).

DIMENSION is an unsigned integer 1..255 specifying the number of dimensions in an array. If DSC.KIND is not DSC_KIND_BIT_ARRAY, then DSC.DIMENSION must be zero.

KIND is the descriptor kind attribute:

DSC_KIND_FIXED_BIT = 81
DSC_KIND_BIT_ARRAY = 82

POINTER is the base address of the bit string or array. This is the quadword-aligned address of the storage containing the first bit of the string or, for an array, the quadword-aligned address of the storage containing element (DSC.LOW_BOUND(1), . . . DSC.LOW_BOUND(n)).

EXTENT specifies the number of bits of contiguous storage occupied by the data.

When DSC.KIND = DSC_KIND_FIXED_BIT, then DSC.EXTENT is an integer $0..2^{64}-1$ specifying the length in bits of the string.

When DSC.KIND = DSC_KIND_BIT_ARRAY, and DSC.DIMENSIONS = 1, and DSC.STRIDE = DSC.ELEMENT_SIZE, then DSC.EXTENT is an integer $0..2^{64}-1$ specifying the length in bits of the storage occupied by the array.

For all other cases, the contents of DSC.EXTENT are *unpredictable*.

ELEMENT_SIZE specifies the number of bits of storage occupied by each element of a bit array, and is present if and *only* if DSC.KIND = DSC_KIND_BIT_ARRAY.

If, and *only* if, DSC.KIND = DSC_KIND_BIT_ARRAY, then the following fields occur once per dimension:

LOW_BOUND(i) is a signed integer specifying the lower bound of the *i*th dimension of the array.

HIGH_BOUND(i) is a signed integer specifying the upper bound of the *i*th dimension of the array.

STRIDE(i) is a signed integer specifying the difference between the bit (not byte) addresses of successive elements of the *i*th dimension of the array.

Run Time Conventions for Bit-addressable Arguments Descriptors

The conventions controlling the use of bit-addressable arguments descriptors are determined by the kind of the descriptor.

- DSC_KIND_FIXED_BIT

  A fixed bit string is sequence of $0..2^{64}-1$ bits with a fixed extent. DSC.EXTENT specifies the number of bits occupied by the string.

  When a fixed bit string is written, the contents of the descriptor must not be modified.

  The absolute bit address of the first bit of a bit string is:

  bit_address = POINTER*8 + POSITION

  The absolute bit address may require more than 64 bits to represent.

The DSC.POINTER-relative bit offset of the last bit in a bit string can always be represented as a signed 64-bit integer $\leq 2^{63} - 1$. This means that the extent passed in a bit string descriptor array must be $\leq 2^{63} - 64$.

- DSC_KIND_BIT_ARRAY

  A bit-addressable array is a ordered sequence of elements, where all elements of any array must be the same ALPHA data type, and therefore have the same length in bits.

  All elements of an array must reside in distinct storage. That is, an ALPHA-64 array descriptor must not specify that any array elements overlap one another.

  When an array is written, the contents of the descriptor must not be modified.

  The absolute bit address of the first bit of a bit array is:

  bit_address = POINTER*8 + POSITION

  The absolute bit address may require more than 64 bits to represent.

  The absolute bit address of element (i,j,k) of a bit array can be computed as follows.

  bit_address = POINTER*8 + POSITION +

  (i –LOW_BOUND(1))*STRIDE(1) +
  (j –LOW_BOUND(2))*STRIDE(2) +
  (k –LOW_BOUND(3))*STRIDE(3)

  or

  bit_address = POINTER*8 + POSITION +

  i*STRIDE(1) + j*STRIDE(2) + k* STRIDE(3) –
  (LOW_BOUND(1)*STRIDE(1) +
  LOW_BOUND(2)*STRIDE(2) +
  LOW_BOUND(3)*STRIDE(3))

  The DSC.POINTER-relative bit offset of the last bit in the last element in a bit array must be $\leq 2^{64} - 1$. This means that the difference between absolute bit addresses of the first and last bits in an array must be $\leq 2^{64} - 64$.

Interchangeability of Fixed Text and Array Descriptors

Array and fixed string descriptors are interchangeable for use in passing contiguous one-dimensional sequences of bits or characters.

For any standard interface that accepts a fixed text string or a contiguous one-dimensional array of characters, either of the following descriptors may be passed:

DSC.KIND = DSC_KIND_FIXED_TEXT
DSC.EXTENT = *n*

or

DSC.KIND = DSC_KIND_BYTE_ARRAY
DSC.EXTENT = *n*
DSC.DIMENSION = 1
DSC.ELEMENT_SIZE = *character size*
DSC.EXTENT = DSC.ELEMENT_SIZE * MAX(0,(DSC.HIGH_BOUND – DSC.LOW_BOUND + 1))

DSC.STRIDE = DSC.ELEMENT_SIZE

For any standard interface that accepts a fixed bit string or a contiguous one-dimensional array of bits, either of the following descriptors may be passed:

DSC.KIND = DSC_KIND_FIXED_BIT
DSC.EXTENT = *n*

or

DSC.KIND = DSC_KIND_BIT_ARRAY
DSC.EXTENT = *n*
DSC.DIMENSION = 1
DSC.ELEMENT_SIZE = 1
DSC.EXTENT = MAX(0,(DSC.HIGH_BOUND – DSC.LOW_BOUND + 1))
DSC.STRIDE = 1

Any called procedure that accepts a fixed text descriptor is allowed to interpret a byte array descriptor as a fixed text descriptor. Likewise, any called procedure that accepts a fixed bit descriptor is allowed to interpret a bit array descriptor as a fixed bit descriptor.

If a called interface requires a one-dimensional character array descriptor, then it must be prepared to accept either type of descriptor and convert a fixed text descriptor to a byte array if necessary. Likewise, if a called interface requires a one-dimensional bit array descriptor, then it must be prepared to accept either type of descriptor and convert a fixed bit descriptor to a bit array if necessary.

RATIONALE:

This approach is designed to satisfy the following goals:

- **unify strings and one-dimensional contiguous arrays;**
- **provide a bulletproof rule for passing strings between languages;**
- **impose no additional programming costs or overhead on users or on string-oriented languages;**
- **impose acceptably small overhead on array-oriented languages;**
- **handle multibyte characters.**

5.2 Data Passing

The fundamental unit of data which gets passed between procedures has been abstracted for purposes of discussion to a concept called an *argument item*. An argument item represents one unit of data being passed.

5.2.1 Argument Passing Mechanisms

The ALPHA calling standard distinguishes three classes of argument items according to the mechanism used to pass the argument:

- Immediate value
- Reference
- Descriptor

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

This standard does not dictate which of the above mechanisms must be used by a given language compiler. Language semantics and/or interoperability considerations may require different mechanisms to be used in any given situation. See Section D.1.2, Data Passing Mechanism, for more discussion of this topic.

Immediate Value

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

Reference

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

Descriptor

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

5.2.2 Argument List Structure

The *argument list* in an ALPHA call is an ordered set of zero or more argument items, which together comprise a *logically* contiguous structure known as the *argument item sequence*. An argument item is represented in 64 bits.

An argument item may be used to pass immediate arguments $\leq$ 64 bits, arguments by reference, and arguments by descriptor. The standard permits any combination of these mechanisms in an argument list.

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 may produce a physically discontiguous argument list. Registers R16..R21 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 which must be located at 0(SP) at the time of the call.

RATIONALE:

The caller needs to have enough scratch registers to compute arguments. In addition, if the caller must make a call to compute some of the arguments, any that are in scratch registers must be saved to memory (but only once - so this is no worse than passing them in memory).

The following table specifies the standard locations in which argument items can be passed.

Table 5–5: Argument Item Locations

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

The general rules that determine the location of any specific argument can be summarized as follows:

1. All argument items are passed in the integer registers or on the stack, *except* argument items that are floating point data passed by immediate value.
2. Floating point data passed by immediate value is passed in the floating point registers or on the stack.
3. Only *one* location in any row in the above table may be used by any given argument list. So, for example, if argument item 3 is an integer passed by value, and argument item 4 is a single precision floating point number passed by value, then argument item 3 is assigned to R18 and argument item 4 is assigned to F19.

The argument list, including both the in-memory portion as well as the portion that is passed in registers, may be read and written by the called procedure. The calling procedure must therefore not make any assumptions about the validity of any part of the argument list after the completion of a call.

5.2.3 Argument Lists and High Level Languages

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

1. Arguments are mapped from left to right to increasing offsets in the argument item sequence. R16 or F16 is allocated to the first argument, and the last quadword of the memory argument list (if any) is allocated to the last argument.
2. Each source language argument corresponds to a single ALPHA calling standard argument item.
3. Each argument item consists of 64 bits.

4. A null or omitted argument, for example CALL SUB(A,,B), is represented by an argument item containing 0.

   No arguments passed by the immediate mechanism may be omitted unless a default value is supplied by the language. (This is to enable called procedures to distinguish an omitted immediate argument from an immediate value argument with the value 0.)

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

5.2.4 Unused Bits In Passed Data

Whenever data is passed by value between two procedures either 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 either sign extended or zero extended as appropriate.

The table below defines the rules for setting or clearing the unused bits.

Key:

- *Sign32* means sign extended to 32 bits - The state of bits <63:32> is unpredictable
- *Sign64* means sign extended to 64 bits
- *Zero32* means zero extended to 32 bits - The state of bits <63:32> is unpredictable
- *Zero64* means zero extended to 64 bits
- *Data32* means data is 32 bits - The state of bits <63:32> is unpredictable
- *Data64* means data is 64 bits
- *Hard* means passed in the layout defined by the Hardware SRM
- *Nostd* means that the state of all high order bits not occupied by the data is unpredictable across a call or return

Table 5–6: Unused Bits in Passed Data

Data Type	Type Designator	Data Size (bytes)	Register Extension Type	Memory Extension Type
byte logical	BU	1	Zero64	Zero64
word logical	WU	2	Zero64	Zero64
longword logical	LU	4	Sign64	Sign64
quadword logical	QU	8	Data64	Data64
byte integer	B	1	Sign64	Sign64
word integer	W	2	Sign64	Sign64
longword integer	L	4	Sign64	Sign64
quadword integer	Q	8	Data64	Data64
F floating	F	4	Hard	Data32
D floating	D	8	Hard	Data64
G floating	G	8	Hard	Data64
F floating complex	FC	2 * 4	Hard	Data64
D floating complex	DC	2 * 8	Hard	N/A
G floating complex	GC	2 * 8	Hard	N/A
IEEE floating basic single S	FS	4	Hard	Data32
IEEE floating basic double T	FT	8	Hard	Data64
IEEE floating basic single S complex	FSC	2 * 4	Hard	Data64
IEEE floating basic double T complex	FTC	2 * 8	Hard	N/A
Small structures of 8 bytes or less	N/A	≤8	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

It is left to the language compilers to map call arguments onto the appropriate data type.

NO data type conversions must be assumed between the calling procedure and the called procedure.

5.2.5 Sending Data

Sending Mechanism

The following represents the rules which govern the allowable mechanisms for sending data.

By immediate value An argument may be passed by immediate value only if

- it is one of the non *complex* scalar data types with size known to be ≤ 64 bits,

  or

- it is a record with size known to be ≤ 64 bits,

  or

- it is a set, implemented as a bit vector, with size known to be ≤ 64 bits,

No form of string, array or complex data type may be passed by immediate value.

A standard immediate argument item must fill 64 bits. This means that unused high-order bits of all data types (including records) must be zero-extended or sign-extended, as appropriate depending on the data type, to fill all unused bits. (See Table 5–6, Unused Bits in Passed Data, for details.)

Large Immediate Arguments Argument values that are larger than 64 bits can be passed by immediate value using *non-standard conventions.*

The most common such convention is to allocate as many fully or partially occupied argument item positions to the argument value as are needed to represent the argument. Partially occupied argument item positions must be zero or sign extended as appropriate to fill the entire item position. The argument count in this case represents the number of argument item positions used rather than the number of actual arguments.

Thus, for example, a 64-bit complex floating point argument could be passed by value in 2 floating point registers. A 26-byte string could be passed by value in four integer registers.

Such large immediate arguments are *not* standard arguments, and should not be used in standard external calls.

By Reference Non-parametric arguments (that is, arguments for which associated information such as string size and array bounds are not required) may be passed by reference in a standard call.

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

Order of Argument Evaluation

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

5.2.6 Receiving Data

When it cannot be determined at compile time if a given in-register argument item is passed in a floating point register or an integer register, the argument information register may be interpreted at runtime to establish where the argument was passed. (See Section 4.5.1, Call Conventions, for details.)

5.2.7 Returning Data

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

- immediate value
- reference
- descriptor

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

Function Value Return By Immediate Value

This section describes the two types of immediate value function return.

RATIONALE:

This approach to function value by immediate value, which allows a common mechanism across both the ALPHA-32 and the ALPHA-64 flavor of the calling standard, is based on the proposition that returning 64 bits is usually the result of returning a 64-bit data item, rather than the result of returning two 32-bit data items, and therefore R0<63:0> is used.

Non-Floating Function Value Return By Immediate Value

A function value is returned by *immediate value* in register R0 if, and *only* if,

- it is a non-floating point scalar data type with size known to be ≤ 64 bits,

  or

- it is a record with size known to be ≤ 64 bits.

  or

- it is an array with size known to be ≤ 64 bits,

  or

- it is a set, implemented as a bit vector, with size known to be ≤ 64 bits.

No form of string may be returned by immediate value.

Two separate 32-bit entities cannot be returned in R0.

A function value < 64 bits returned in R0 must be zero-extended or sign-extended, as appropriate depending on the data type (see Table 5–6, Unused Bits in Passed Data, for more details), to a full quadword.

Floating Function Value Return By Immediate Value

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

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

Function Value Return By Reference

A function value is returned by *reference* if, and *only* if,

- the actual size of the function value is known to both the calling procedure and the called procedure, but the value cannot be returned by immediate value (because the function value requires more than 64 bits, the data type is a string or an array type, etc.)

  and

- the function value can be returned in a contiguous region of storage.

The actual-argument list and the formal-argument list are shifted to the right by one argument item. The new, first argument item is reserved for the function value. This *hidden* first argument is included in the count and register usage information that is passed in the argument information register (see Section 4.5.1, Call Conventions, for details).

The calling procedure must provide the required contiguous storage and pass the address of the storage as the first argument. This address *must* specify storage 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.

Function Value Return By Descriptor

A function value is returned by *Descriptor* if, and *only* if,

- the actual size of the function value is not known to both the calling procedure and the called procedure and the value cannot be returned by immediate value (because the function value requires more than 64 bits, the data type is a string or an array type, etc.)

  and

- the function value can be returned in a contiguous region of storage.

Non-contiguous function values are language-specific and can not be returned as a standard conforming return value.

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

There are two distinct cases where a descriptor return value is used. They are:

- When the caller determines the maximum size of the returned value
- When the called routine determines the maximum size of the returned value

In each case the actual-argument list and the formal-argument list are shifted to the right by one argument item. The new, first argument item is reserved for the function value. This *hidden* first argument is included in the count and register usage information that is passed in the argument information register.

The initialization and usage of the function value return descriptor is environment specific. See Section 5.2.7, Returning Data, for more details.

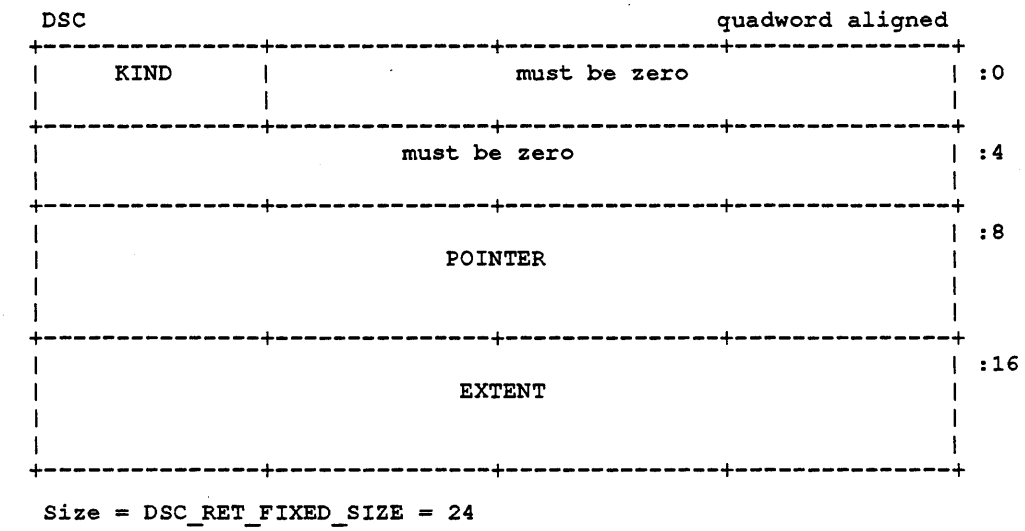
Function Value Return By Descriptor

The two cases of function value return by descriptor are described below. The mechanism to be used is chosen by the calling procedure, and is specified by the kind of the descriptor passed as the first argument item.

Caller Defined Return By Descriptor

When the calling procedure determines the maximum size of the returned value it must provide the required contiguous storage and pass the address of a naturally aligned descriptor that describes that storage as the first argument. This descriptor is laid out as follows:

Figure 5–4: ALPHA-64 Caller Defined Return Value Descriptor Format



The calling routine must initialize the descriptor as follows:

- DSC.KIND must be set to DSC_KIND_FIXED_TEXT
- DSC.POINTER must be set to point to the buffer where the value is to be returned
- DSC.EXTENT = an integer $0..2^{64} - 1$ specifying the length in bytes of the storage

To return the value the called function must:

- Write the return value into the storage specified by the hidden descriptor argument, truncating the return value if its length exceeds the value of DSC.EXTENT.
- Provide whatever padding is required by the semantics of the language of the called function.

```
\\
It should be noted that various languages have different language requirements.
FORTRAN and PASCAL, for example, require BLANK padding while Ada does
not pad at all.

This standard makes no attempt to reconcile these differences.
\\
```

The descriptor may be allocated in read-only storage by the caller and thus the called function must never modify the descriptor.

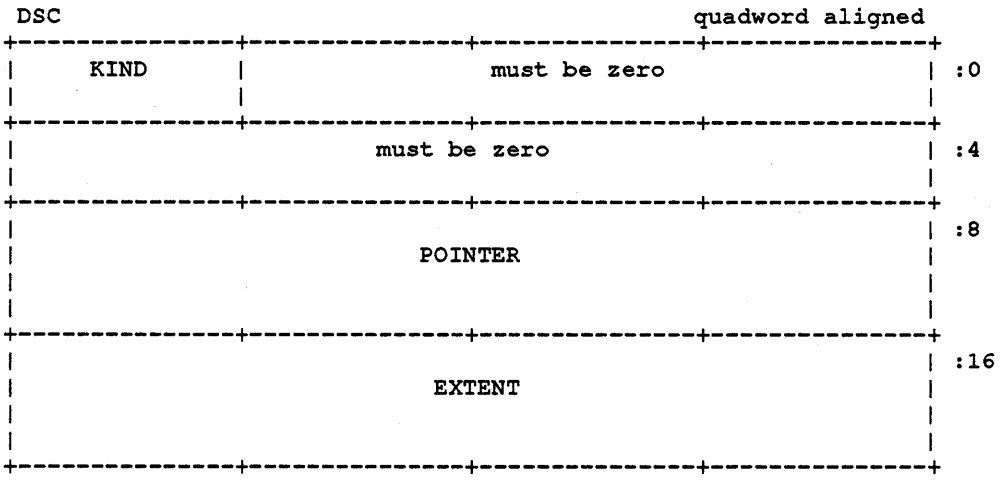
RATIONALE:

This type of function value return is specifically for FORTRAN.

Called Routine Defined Return By Descriptor

When the called routine determines the maximum size of the returned value the stack is used to return the actual data of the value. In this case the descriptor pointed to by the hidden first argument is used as follows:

Figure 5–5: ALPHA-64 Called Routine Defined Return Value Descriptor Format



The caller must pass as the first argument a function return descriptor initialized as follows:

DSC.KIND = DSC_KIND_STACK_RETURN = 130
DSC.POINTER = *unpredictable*
DSC.EXTENT = 0

The descriptor must be allocated in writeable storage, and will be modified by the called function.

The called function must return its value as follows:

1. Allocate stack storage sufficient to contain the return value and place the return value on the stack.
2. Update the descriptor as follows:

   DSC.KIND = DSC_KIND_STACK_RETURN
   DSC.POINTER = the address of the first byte of the return value
   DSC.EXTENT = an integer $0..2^{64} - 1$ specifying the length in bytes of the return value

   NOTE

   In all cases DSC.POINTER must be set before DSC.EXTENT is made non zero. This allows unwind operations to properly restore the stack pointer.

3. The return code sequence must not reset SP such that any part of the function return value is contained by a stack address lower than 0(SP). The return value must be entirely contained by stack storage at or above 0(SP).

When control returns to the calling procedure, the contents of register R0 are *unpredictable*. The caller must manage the return value and SP. The caller may copy the return value from the stack to some other storage (possibly to a higher address on the stack) and reset SP appropriately to reflect the return from the called function.

Run Time Conventions for Function Value Return Descriptors

Any standard-conforming function that supports returning values using the DSC_KIND_STACK_RETURN type of descriptor must also be prepared to handle the other descriptor return value mechanism, and must return its value according to the mechanism chosen by the caller.

Functions that may return a stack function return value *must* use the following algorithm to determine which of the two mechanisms to use:

1. If DSC.KIND = DSC_KIND_FIXED_TEXT, then use the fixed buffer mechanism.
2. Otherwise, use the stack return mechanism.

It is important that functions do not specifically test for DSC.KIND = DSC_KIND_STACK_RETURN. This is because future extensions to this standard may specify new function return mechanisms that define new KIND codes, but that are upward compatible with the stack return mechanism specified here.

```
\\
The case in mind here is return via heap storage allocated by the called
function, in which case the caller could "request" return in such heap storage,
but a called function could at its own option override that request and use the
stack return mechanism instead, by changing DSC.KIND in the passed descriptor.
Callers that "request" return in heap storage would have to be prepared to
handle either heap or stack return from the called function.
\\
```

Any function that tests for DSC.KIND = DSC_KIND_STACK_RETURN may not be upward compatible with future versions of this standard.

For the same reason, functions that return a value by the stack return mechanism must *always* set DSC.KIND = DSC_KIND_STACK_RETURN. Any function that returns a value by stack return mechanism and does not set DSC.KIND = DSC_KIND_STACK_RETURN may not be upward compatible with future versions of this standard.

RATIONALE:

It is assumed that there will be an OTS routine that language compilers will call that might look like this:

```
;
;       Load descriptor address in R0
;       Load SP in R1
;       Load string address in R22
;       Load string length in R23
;

        JSR     OTS$DYN_RETURN

;       Returns string length in R0
;       Returns SP to use in R1

        MOV     R1,SP
        RET     R31,(R26)
```

5.3 Static Data

5.3.1 Alignment

On the ALPHA architecture, memory references to data that is not naturally aligned may result in alignment faults, which can *severely* degrade the performance of all procedures that reference the unnaturally aligned data.

For this reason, arguments and function return values on ALPHA systems should be naturally aligned. For example, 8-bit character strings should start on byte boundaries; 16-bit integers should start at addresses that are a multiple of at least 2 (word alignment); single precision real values should start at addresses that are a multiple of at least 4 (longword alignment); double precision real values should start at addresses that are a multiple of at least 8 (quadword alignment); and so forth.

Data types larger than 64 bits should use 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 of which the aggregate is composed. The alignment requirement of an aggregate is that all elements of the aggregate be naturally aligned. Varying 8-bit character strings must, for example, start at addresses that are a multiple of at least 2 (word alignment) because of the 16-bit count at the beginning of the string; 32-bit integer arrays start at a longword boundary, irrespective of the extent of the array.

5.3.2 Record Format Conventions

The ALPHA calling standard record layout rules are designed to provide good run time performance on all implementations of the ALPHA architecture, and to provide the required level of compatibility with conventional VAX/VMS operating environments.

This standard therefore defines two record layout conventions:

- record layout conventions optimized for the ALPHA architecture, referred to as *ALPHA preferred* record layouts
- record layout conventions that are compatible with those traditionally used by VAX/VMS languages, referred to as *VAX/VMS compatible* record layouts.

NOTE

Although compiler implementors must make appropriate business decisions, it is STRONGLY advised that all ALPHA high level language compilers should support both record layouts.

Only these two record layouts may be used across standard interfaces or between languages. Languages may support other language-specific record layout conventions, but such other record layouts are not standard.

The ALPHA preferred record layout conventions should be used unless interchange is required with conventional VAX applications that use the VAX/VMS compatible record layouts.

ALPHA Preferred Record Layout

The ALPHA preferred record layout conventions ensure that:

- all components of a record or subrecord are naturally aligned
- the layout and alignment of record elements and subrecords is independent of any record or subrecord in which they may be embedded
- the layout and alignment of a subrecord is the same as if it were a top level record
- declaration in high level languages of standard records for interlanguage use is reasonably straightforward and obvious, and meets the requirements for source level compatibility between ALPHA and VAX/VMS languages

The ALPHA preferred record layout is defined by the following conventions:

1. 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.
2. The first bit of a record or subrecord must be directly addressable; i.e. it must be byte aligned.
3. Records and subrecords must be aligned according to the largest natural alignment requirements of the contained elements and subrecords.
4. 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.
5. All other components of a record must start at the next available naturally aligned address for the data type.
6. Strings and arrays must be aligned according to the natural alignment requirements of the data type of which the string or array is composed.
7. The length of an array element is a multiple of its alignment, even if this leaves unused space at its end. The length of the whole array is the sum of the lengths of its elements.

VAX/VMS Compatible Record Layout

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

1. 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.
2. 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.
3. All other components of a record must start at the next available byte in the record, and any unused bits following the last-used bit in the last-used byte of each component must be filled out to the next byte boundary such that any following data starts on a byte boundary.
4. Subrecords must be aligned according to the largest alignment of the contained elements and subrecords. This means that a subrecord always starts at the next available byte unless it consists entirely of unaligned bit data and it immediately follows an unaligned bit string, unaligned bit array, or a subrecord consisting entirely of unaligned bit data.
5. Records must be aligned on byte boundaries.

CHAPTER 6

EVENT PROCESSING

This chapter discusses specifications related to events that are normally outside the normal program flow.

6.1 Exception Handling

This section on exception handling discusses the considerations involved in the notification and handling of exceptional events during the course of normal program execution.

Exception notification and handling is closely tied to the hardware mechanisms for exceptional event reporting. The ALPHA architecture is notably different from a conventional VAX in this area.

```
\\
The basic designs and mechanisms of ALPHA Exception handling and unwinding are
modeled after the VAX/VMS design with extensions and corrections to support
POSIX and multithread environments better. That design is defined in Sections
"VAX Conditions", "Operations Involving Condition Handlers", "Properties of
Condition Handlers", and "Multiple Active Signals" of the VAX
Procedure Calling and Condition Handling Standard, Revision 10.3 which may
serve as useful background reading.
\\
```

6.1.1 Exception Handling Requirements

The exception handling capabilities specified in this standard are for support of:

- Reliable programmer and program control over response to exceptions and reporting of such exceptions, and over the flow of control when exceptions occur.
- Capability for subsystems and applications to override system messages to provide a more suitable application oriented interface, specifically including modular, multinational message and error reporting.
- Orderly termination of layered applications.
- Correct and predictable exception handling in a multilanguage environment
- Support for the construction of modular, maintainable multilanguage applications.
- Support for parallel multithreaded application execution, including
  - **a.** Per-thread exception handling.
  - **b.** Handling of asynchronous exceptions.
  - **c.** Safe thread exit in a multithreaded environment.

- Support for at least the minimal exception handling requirements of the Concert Multithread Architecture including support:
  - To raise a particular kind of exception to denote that a particular error has occurred in the procedure invocation associated with a particular call frame.
  - To search the call chain of the thread receiving the exception notification (from the most recently invoked to the least recently invoked procedure) to locate a handler for the exception.
  - For such a handler to be able to resume thread execution at a program location designated by the handler, which requires terminating all intervening frames in the call chain.
- Procedure based exception handling must be capable of coexisting and interoperating with POSIX-defined signal handling. The interoperation of these facilities, and the limits of such interoperation, is described in this standard.

6.1.2 Exception Handling Overview

Basic workings

When an exception occurs (is *raised*), the normal flow of control in the current thread is interrupted, the context saved, and control is transferred to the exception handling support code. This support code marshals 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 nature of the exception, the environment within which the exception was raised, and 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 may respond to the exception in several ways, including various combinations of the following:

- Perform some action that affects the context of the thread (possibly 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 in one of the following ways:

- Indicate that the exception handling support code should *Reraise* the exception and resume the search for another handler.
- Indicate that the exception handling support code should *Continue* execution of the interrupted thread at the location indicated by the saved Exception Program Counter.

- *Unwind*, which causes the exception handling support code to resume execution of the thread at a point different than the point at which it was interrupted, or terminate the execution of the thread.

Exceptions

There are three basic kinds of exceptions:

- Those caused by an unwind operation (called *unwind exceptions*)
- Those caused in support of the POSIX 1003.1 *signal* function (called *signal exceptions*)
- Those caused by other software or hardware notification mechanisms (called *general exceptions*)

Unwind Exceptions

An *unwind exception* results from the invocation of the unwind support code by a thread, and is always delivered to the thread which invoked the unwind.

Unwind exceptions are delivered as part of the notification process that an unwind is in progress (see Section 6.2, Unwinding, for details).

Signal Exceptions

A signal exception results from the delivered of a POSIX signal which is subsequently marshaled into an exception that can be handled using the capabilities defined by this standard. Signal exceptions may only be raised in POSIX-conforming environments.

General Exceptions

General exceptions may be further categorized as one of:

Software Caused

A software caused general exception is raised as the result of the invocation of an exception raising procedure and is always delivered to the thread which made the call.

Such an exception may be raised at any point during thread execution. Applications and language run time libraries may 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 may be raised as general exceptions.

Hardware Caused

A *hardware exception* occurs when a thread performs some action which causes an exceptional state to exist in the hardware. Such a state will cause the currently active thread to be interrupted and the exception handling support code to be invoked. This invocation constitutes the raising of a hardware caused general exception.

A hardware caused general exception is always delivered to the thread that executed the instruction which caused the exception.

Exactly which hardware events can result in an exception, the state of the machine when a hardware exception occurs, the interpretation of the exception-related information which is delivered to a user mode thread, and circumstances under which execution can be continued are specific to individual hardware exceptions. Hardware exceptions are fully defined by the *ALPHA System Reference Manual* which should be consulted for additional information.

Exception Properties

All exceptions are handled with the same interfaces, data structures, and algorithms. That is, there is unified exception handling for all kinds of exceptions, regardless of their origination.

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

6.1.3 Data Structures for Exception Handling

Exception Record

The fundamental data structure for describing exceptions is the *exception record*.

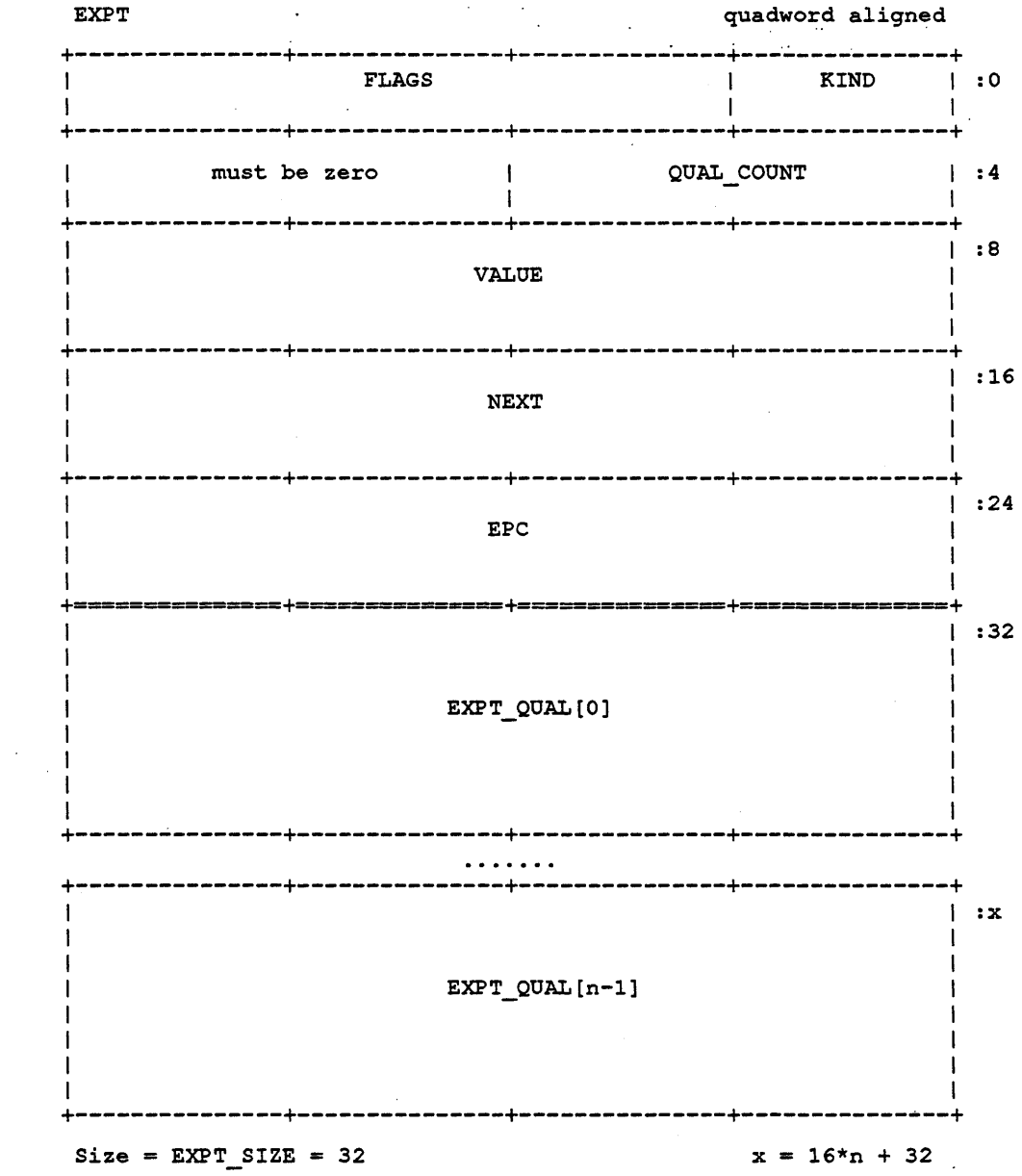
Exception records can form a linked list. Each record in a list describes one exception.

The first exception record in the list describes the *primary exception*. Additional *secondary exceptions* may be specified by additional exception records in the list. Secondary exceptions qualify or elaborate the primary exception; they may be raised at the same time as the primary exception, or a handler may add new secondary exceptions to the list before handling or reraising the exception.

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

Exception records are defined as follows:

Figure 6–1: Exception Record Format



EXPT.KIND specifies the kind of the exception record. This field is significant only in the primary exception record; its contents are *unpredictable* in secondary exception records.

Valid EXPT.KIND values are defined as follows:

- If the exception record is an unwind exception record then EXPT.KIND contains EXC_DEC_UNWIND.
- If the exception record is a general exception record then EXPT.KIND contains EXC_DEC_RAISE.
- Otherwise, EXPT.KIND contains a POSIX 1003.1 style signal number that best approximates the exception which is represented by the exception record.

EXPT.FLAGS is a bitvector of flags which further qualify the exception. These flag bits are significant only in the primary exception record; their state is *unpredictable* in secondary exception records. EXPT.FLAGS bits are logically divided into two groups. The first group (called detail flags) provide additional information about the exception. The second group (called environment flags) provide additional information about the environment in which the exception is being delivered.

Valid EXPT.FLAGS bits that give additional detail are:

- EXPT.FLAGS<0> is denoted by IS_NONRESUMABLE

  If IS_NONRESUMABLE is 1, the exception handler must not return STATUS_CONTINUE
- EXPT.FLAGS<1> is denoted by IS_EXIT_UNWIND

  IS_EXIT_UNWIND is 1 if the exception handler is being invoked because of an unwind operation that will terminate execution of the thread.

  IS_EXIT_UNWIND must be 0 unless EXPT.KIND = EXC_DEC_UNWIND.
- EXPT.FLAGS<2> is denoted by IS_LONGJMP.

  IS_LONGJMP is 1 if the exception handler is being invoked because of a general unwind operation with the semantics of *longjmp()*. The handler *must not* modify the exception record, and *must not* raise an exception or an overlapping unwind.

  IS_LONGJMP must be 0 unless EXPT.KIND = EXC_DEC_UNWIND.

  IS_LONGJMP is specific to POSIX-conforming environments. This flag bit is significant only in the primary exception record within POSIX-conforming environments; its state is *unpredictable* in secondary exception records and in environments that are not POSIX-conforming.
- EXPT.FLAGS<7:3> must be 0.

Valid EXPT.FLAGS bits that give additional information about the environment at the time of exception delivery are:

- EXPT.FLAGS<8> is denoted by IS_UNWIND_INPROG

  IS_UNWIND_INPROG is 1 if an unwind is in progress at the time this exception is delivered.
- EXPT.FLAGS<9> is denoted by IS_SIGNAL_INPROG

  IS_SIGNAL_INPROG is 1 if a signal is in progress at the time this exception is delivered.
- EXPT.FLAGS<10> is denoted by IS_ASI_INPROG

  IS_ASI_INPROG is 1 if an asynchronous software interrupt is active at the time this exception is delivered.
- EXPT.FLAGS<11> is denoted by IS_PRIMARY

  IS_PRIMARY is 1 if this handler is being invoked as a primary handler.

  RATIONALE:

  If a handler is used in multiple places, this bit allows the handler to know why it is being invoked. Note that there is no VMS style *depth* mechanism to do this otherwise.
- EXPT.FLAGS<12> is denoted by IS_LAST_CHANCE

IS_LAST_CHANCE is 1 if this handler is being invoked as a last-chance handler.

- EXPT.FLAGS<13> is denoted by IS_DURING_ASYNC

  IS_DURING_ASYNC is 1 if an asynchronous software exception handler has reraised a nested exception or has been unwound. This indicates that a nested exception was raised or an unwind was initiated during asynchronous exception handling, which was not handled within the scope of the exception handler.

- EXPT.FLAGS<14> is denoted by IS_DURING_VECTORED

  IS_DURING_VECTORED is 1 if a vectored handler has reraised a nested exception or has been unwound. This indicates that a nested exception was raised or an unwind was initiated during the execution of a vectored handler, which was not handled within the scope of that handler.

- EXPT.FLAGS<23:15> must be 0.

EXPT.QUAL_COUNT is the unsigned number of exception-specific qualifiers in the exception record.

EXPT.VALUE is a quadword value denoting the exception. The actual value for a given exception type is *system defined*.

EXPT.NEXT is either zero or is the address of the next exception record in the list.

EXPT.EPC is the continuation program counter for the exception.

For hardware exceptions, asynchronous software exceptions, and signal exceptions, this is the address of the instruction following that at which the hardware exception, asynchronous exception, or signal interrupted execution of the thread.

For synchronous software exceptions and unwind exceptions, this is the address of the instruction immediately following the call to the library routine that raised the exception or invoked the unwind, respectively.

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

Each **EXPT.EXPT_QUAL(n)** is two quadwords which provide additional information specific to the exception, and may contain information intended for display in messages.

Exception Qualifiers

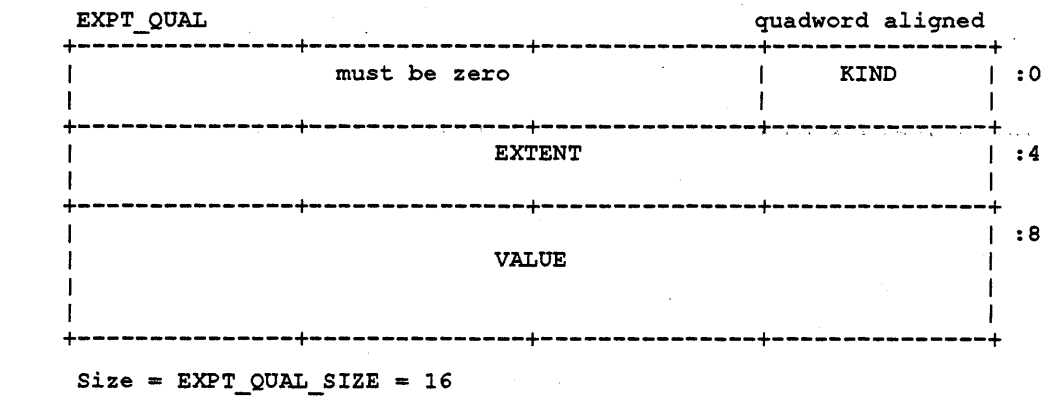
There are two forms of exception qualifiers. They are:

- Immediate exception qualifiers
- Indirect exception qualifiers

Immediate Exception Qualifiers

Immediate exception qualifiers contain values that require eight bytes of storage or less.

Figure 6–2: Immediate Exception Qualifier Format



EXPT_QUAL.KIND is QUAL_KIND_IMMEDIATE.

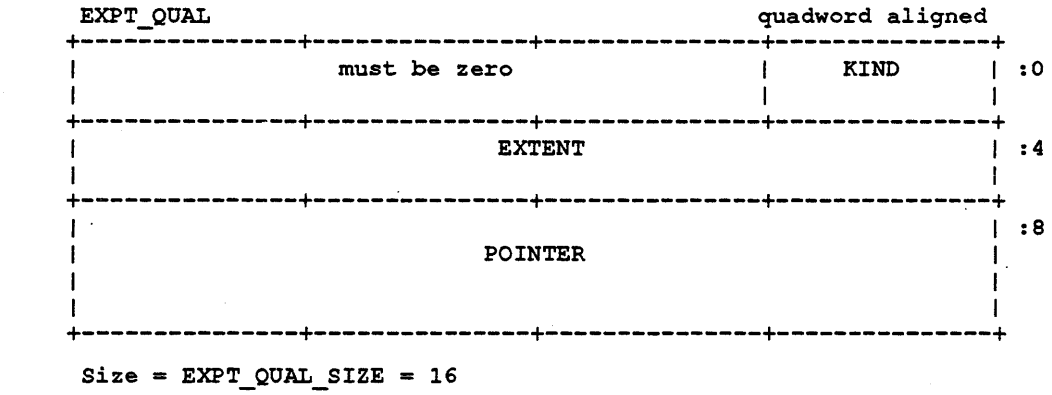
EXPT_QUAL.EXTENT specifies the number of bytes of contiguous storage occupied by the qualifier value. The maximum value of EXPT_QUAL.EXTENT for this kind of exception qualifier is 8.

EXPT_QUAL.VALUE contain the value of the qualifier. The qualifier value must be stored starting at byte 8.

Indirect Exception Qualifiers

Indirect exception qualifiers specify values that require more than eight bytes of storage.

Figure 6–3: Indirect Exception Qualifier Format



EXPT_QUAL.KIND is QUAL_KIND_INDIRECT.

EXPT_QUAL.EXTENT specifies the number of bytes of contiguous storage occupied by the qualifier data.

EXPT_QUAL.POINTER contains the absolute address of the first byte of storage occupied by the data.

Exception Records for Unwind Exceptions

Unwind exceptions are characterized by having EXPT.KIND = EXC_DEC_UNWIND. The type of unwind operation is further qualified by the detail flags in EXPT.FLAGS which specify if the unwind operation is part of a *longjmp()* invocation or is the result of a thread exit.

The reason code for the unwind as well as any supplied qualifiers is represented in EXPT.VALUE in the same way as is done for general exceptions.

Exception Records for Signal Exceptions

Signal exceptions are characterized by having EXPT.KIND ≠ EXC_DEC_RAISE or EXC_DEC_UNWIND.

In exception records for signal exceptions, the value of EXPT.VALUE is the signal number, and the first exception qualifier (if present) is a parameter which further qualifies the signal. This parameter (sometimes referred to as a *signal code* as opposed to a *signal number* is most useful for hardware generated signals. For instance, in a POSIX conforming environment, all arithmetic exceptions are delivered with a signal number of SIGFPE. This *signal code* can further qualify this exception by indicating that it was a floating underflow, integer overflow, or other arithmetic exception.

```
\\
In a POSIX conformant environment there is no mechanism provided for
a user program to deliver a signal with a qualifier of any kind. Thus,
when this qualifier is present it is because it was produced by some
system facility (usually the hardware exception dispatching code).
\\
```

Exception Records for General Exceptions

General exceptions are characterized by having EXPT.KIND = EXC_DEC_RAISE.

Software Caused Exceptions

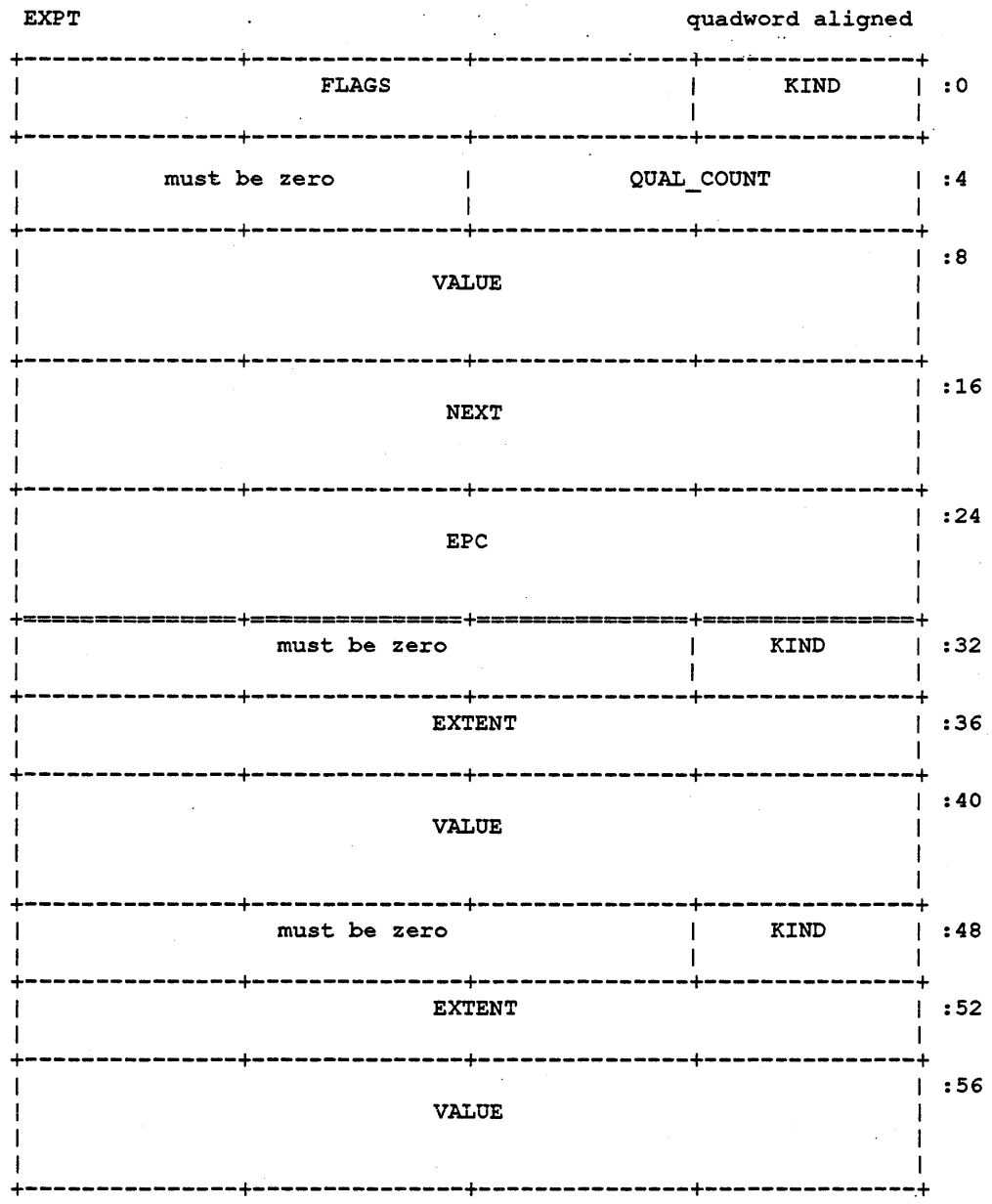
The information in exception records for general and unwind exceptions may vary widely from a simple single exception to a long chain of exceptions and exception qualifiers. This specification defines the conventions for constructing these exception records (see Section 6.1.3 above). A complete enumeration of all possible combinations is beyond the scope of this document.

Hardware Caused Exceptions

All ALPHA hardware exceptions have exception information associated with them. This information may be as little as the exception type and exception PC or as much as two additional registers worth of information. The specific information that is supplied with each exception type is defined by the ALPHA architecture SRM.

When a hardware exception is raised, the exception record passed to exception handlers includes the exception information supplied by the hardware. This information is converted to exception argument format and marshaled into the exception record to produce an exception record as follows:

Figure 6–4: Hardware Caused Exception Record



EXPT.KIND is EXC_DEC_RAISE or a POSIX 1003.1 style signal number depending on how this exception is being delivered.

EXPT.FLAGS is as appropriate (see above).

EXPT.QUAL_COUNT is 2.

EXPT.VALUE is the type of hardware exception.

EXPT.NEXT is 0.

EXPT.EPC is the PC that was saved on the stack by the hardware when the exception processing code was invoked.

EXPT_QUAL[0].KIND is QUAL_KIND_IMMEDIATE.

EXPT_QUAL[0].EXTENT is 8.

EXPT_QUAL[0].VALUE is the contents of R4 as passed by the hardware.

EXPT_QUAL[1].KIND is QUAL_KIND_IMMEDIATE.

EXPT_QUAL[1].EXTENT is 8.

EXPT_QUAL[1].VALUE is the contents of R5 as passed by the hardware.

Each hardware exception has specific R4 and R5 values, as defined by *ALPHA System Reference Manual* which should be consulted for further information.

6.1.4 Exception Handlers

Exception Handler Types

There are four types of exception handlers:

- *Primary* handlers
- *Frame-based* handlers
- *Last-chance* handlers
- The system *catchall* handler

Primary Handlers

Primary handlers may only be established at run time, and are independent of the procedure stack frame structure of the executing thread. Primary handlers are normally utilized to provide language independent services such as debugging, instrumentation, and exception processing by facilities that might not have an active procedure on the call stack when an exception occurs.

When an exception is raised, the exception handling support code searches for primary handlers before any others. No other types of exception handlers are invoked unless all primary handlers have been invoked and have reraised the exception.

Frame-based Handlers

A frame-based handler is established when a procedure whose descriptor specifies an exception handler becomes current. Thus, frame-based handlers are usually bound to a procedure at compile time, and are located at run time via the procedure descriptor.

These exception handlers are normally used to implement a particular language's exception handling semantics.

If all primary handlers reraise an exception, the exception dispatcher searches for frame-based handlers. The frame-based handlers which may be invoked are those established by active procedures, from the most current procedure to the oldest predecessor.

Last-Chance Handlers

Like primary handlers, last-chance handlers may only be established at run time. If all frame-based handlers reraise an exception the exception dispatcher will invoke the last-chance handlers.

Like primary handlers, last-chance handlers are normally utilized to provide language independent services such as debugging and instrumentation.

System Catchall Handler

If all other exception handlers reraise an exception, the system catchall handler is invoked.

The system catchall handler is not established by a thread. It is supplied by the system, is always established, and is always valid.

The action of the system catchall handler is *undefined* by this standard. It may produce an error message, continue thread execution, terminate execution of the thread in which the exception was raised, or any other system dependent action.

General properties

A thread may have an arbitrary number of primary and last-chance handlers simultaneously established.

Primary Handlers

Primary handlers are invoked in FIFO order with respect to the order in which they were established.

Last-Chance Handlers

Last-chance handlers and are independent of the procedure stack frame structure of the executing thread.

Last-chance handlers are invoked in LIFO order with respect to the order in which they were established.

Other Properties of Exception Handlers

Handling Exceptions

An exception handler that conforms to this standard may not handle any exception that it 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.

Access to Memory

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

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

6.1.5 Establishing Handlers

Establishing Primary Handlers

A thread may establish a primary handler, to be called after all previously established primary handlers have reraised an exception, via a system library function defined as follows:

LIB_ESTABLISH_PRIMARY(HANDLER, DATA)

Arguments:

HANDLER	The procedure value of the exception handler to be established as a primary handler.
DATA	A 64-bit establishment-specific value to be passed to the handler. (This corresponds to the handler data that is stored in the procedure descriptor for frame based exception handlers)

Function Value:

ESTAB_HANDLE	A 64-bit value which may be used to reference this establishment of a handler to revoke it. If the return value is ESTAB_HANDLE_NULL, then an error has occurred and the procedure was not established as a handler

Establishing Frame-Based Handlers

The list of established frame-based handlers for a thread is defined by the thread's procedure invocation chain.

A procedure descriptor for which PDSC.FLAGS.HANDLER_VALID is 1 must specify in PDSC.STACK_HANDLER or PDSC.REG_HANDLER (as appropriate) the self-relative offset to the procedure descriptor of an exception handler. The exception handler specified by an invocation descriptor is established when that descriptor is added to the invocation chain (that is, when the procedure designated by the descriptor becomes current), remains established as long as that procedure invocation is part of the invocation chain, and is revoked when that descriptor 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 which is established at any moment is defined by the current procedure call structure.

Dynamic activation and deactivation of exception handlers is not defined by this calling standard (and in fact 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 which choose to support this functionality may set up language-specific static exception handlers that provide the dynamic exception handling semantics of that language. These 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 (i.e. calls to any *LIB$ESTABLISH* type dynamic activation routines). This requirement stems from the need to add appropriate DRAINT instructions and other compile time considerations needed to make dynamic exception handling function correctly.

```
\\
There may be additional protocols and conventions for dynamic exception
handling. These may be needed, for example, to enable a debugger to do a good
job within the language exception handling environment.  These conventions will
be driven by the requirements of the languages and the language support
utilities, and will not be addressed by the calling standard.
\\
```

Establishing Last-Chance Handlers

A thread may establish a last-chance handler, to be called before any previously established last-chance handlers are called, via a system library function defined as follows:

LIB_ESTABLISH_LAST_CHANCE(HANDLER, DATA)

Arguments:

HANDLER	The address of the procedure descriptor of the exception handler to be established as a last-chance handler.
DATA	A 64-bit establishment-specific value to be passed to the handler. (This corresponds to the handler data that is stored in the procedure descriptor for frame based exception handlers)

Function Value:

ESTAB_HANDLE	A 64-bit value which may be used to reference this establishment of a handler to revoke it. If the return value is ESTAB_HANDLE_NULL, then an error has occurred and the procedure was not established as a handler

The System Catchall Handler

The system catchall handler can not be established or revoked by any code which conforms to this standard. This exception handler is always established and has *system defined* effects.

6.1.6 Revoking Exception Handlers

A thread may revoke a previously established primary or last-chance handler by calling a system library function defined as follows:

LIB_DISESTABLISH_HANDLER(ESTAB_HANDLE)

Arguments:

ESTAB_HANDLE	A value previously returned by *LIB_ESTABLISH_PRIMARY()* or *LIB_ESTABLISH_LAST_CHANCE()*, specifying the vectored handler to be revoked.

Function Value:

STATUS	A status value: STATUS_SUCCESS indicates that the specified handler was revoked. STATUS_FAILURE indicates that the argument did not specify an established vectored handler.

6.1.7 Raising Exceptions

Raising Unwind Exceptions

The mechanism used to raise an unwind exception vary with the intended resulting action to be performed. See Section 6.2, Unwinding, for details.

Raising Signal Exceptions

```
\\
Issue: This assumes a connection between signals and ehf
\\
```

Signal exceptions may 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 notifications such as the POSIX defined *alarm()* function. Complete exposition of these circumstances is beyond the scope of this document.

A variety of run time support packages may chose the signal mechanism as a means to provide certain features. One such package is the ANSI C standard support package which provides the *raise()* function to enable a programmer raise a synchronous signal exception. (defined as *raise(int sig)* - see *Draft Proposed American National Standard for Information Systems Programming Language C* section 4.7.2.1 for details) Another such package is the POSIX support package which provides the *kill()*. The basic parameters of this function are provided below as an example of one means of raising a signal exception.

A thread may raise a signal exception in its own or another process's context by calling a POSIX library routine defined as follows:

kill(pid, sig)

Arguments:

pid_t pid;	The process identification of the process to receive the signal.
int sig;	The signal value to be delivered.

Function Value:

0	Success
-1	Failure

```
\\
The above is presented simply to convey the basic ideas of raising a
signal. IEEE standard 1003.1-1988 section 3.3.2 should be referenced
for complete details of the workings of this call.
\\
```

When raising signal exceptions, EXPT.EPC and EXPT.KIND are ignored in the primary exception record passed in the call. *kill()* sets EXPT.EPC to the return address of the invoking call, and sets EXPT.KIND to the appropriate POSIX 1003.1 style signal number.

Raising General Exceptions

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

LIB_RAISE(EXPT_REC_ADDR)

Arguments:

EXPT_REC_ADDR	The address of a primary exception record.

Function Value:

None.

When raising general exceptions, EXPT.EPC and EXPT.KIND are ignored in the primary exception record passed in the call. *LIB_RAISE()* sets EXPT.EPC to the return address of the invoking call, and sets EXPT.KIND to EXC_DEC_RAISE.

If *LIB_RAISE()* detects that the exception record passed via the first argument is not a valid exception record, it raises the exception STATUS_INVALID_EXCEPTION.

6.1.8 Invocation of Exception Handlers

Order of Invocation

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

All primary handlers are first invoked in FIFO order with respect to the order in which they were established.

If no primary handlers have been established, or if all reraise the exception, then any frame-based handlers are invoked in order from that established by the most current procedure invocation to the oldest predecessor in the invocation chain.

If no frame-based handlers have been established, or if all reraise the exception, then the last-chance handlers are invoked in LIFO order with respect to the order in which they were established.

Finally, if no other exception handlers have been established, or if all reraise the exception, then the system catchall handler is invoked.

Nested Exceptions

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, is as follows:

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.

   This is the same invocation as item 1 (that in which the nested exception was raised) if there are zero invocations in item 2.
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 established handlers of which were invoked for the exception that immediately preceded the nested exception, and all of which reraised.
6. The establisher of the most current handler.

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

Established handlers are invoked in reverse order with respect to that in which their establishers were invoked; that is, the search for procedure invocations which have established handlers is in order from 1 to 7.

Any handlers established by the invocations described by items 1, 2, 3, and 7 are always invoked for a nested exception.

Any handlers established by the invocations described by items 4, 5, and 6 are invoked if, and *only* if, the procedure descriptor for the establisher flags that handler as reinvokable. This means that no handler will be invoked which has already been invoked for an active exception unless that handler is flagged as reinvokable. In particular, this applies to any handler established by a descendent of the establisher of the most current active handler.

If further nested exceptions occur, this procedure invocation chain structure is repeated for those further nested exceptions, and frame-based handlers are invoked according to the above rules, in order from those established by the most current procedure to those established by the oldest predecessor. No handler will be invoked which has already been invoked for an active exception unless that handler is flagged as reinvokable, and no handler established by a descendent of the establisher of any active handler will be invoked unless it is reinvokable.

Steps for Locating and Invoking Handlers for Exceptions

When an exception is raised, the steps that implement the above explanation are as follows:

1. Locate the first-established primary handler, if any.
2. If no established primary handler was located, go to step 7.
3. Invoke the vectored handler just located.

4. If the handler returns STATUS_CONTINUE or initiates an unwind, exit these steps.
5. Locate the next-established primary handler, if any.
6. Go to step 2.
7. Let *current_invocation* be the procedure invocation in which the exception was raised.
8. If *current_invocation* does not establish a handler, go to step 19.
9. Invoke the handler established by *current_invocation*.
10. If the handler returns STATUS_CONTINUE or initiates an unwind, exit these steps.
11. If *current_invocation* is not itself an active handler, go to step 19.
12. Locate the establisher of *current_invocation*.
13. Let *current_invocation* be the procedure invocation which invoked *current_invocation*.
14. If the *current_invocation* does not establish a handler, go to step 18.
15. If the handler is not flagged as reinvokable by its establisher, go to step 18.
16. Invoke the handler established by *current_invocation*.
17. If the handler returns STATUS_CONTINUE or initiates an unwind, exit these steps.
18. If *current_invocation* is not the establisher located in step 12, go to step 13.
19. If *current_invocation* is the beginning of the procedure invocation chain, go to step 22.
20. Let *current_invocation* be the procedure invocation which invoked *current_invocation*.
21. Go to step 8.
22. Locate the last-established last-chance handler, if any.
23. If no established last-chance handler was located, go to step 28.
24. Invoke the vectored handler just located.
25. If the handler returns STATUS_CONTINUE or initiates an unwind, exit these steps.
26. Locate the previously-established last-chance handler, if any.
27. Go to step 23.
28. Invoke the system catchall handler.
29. Force thread to terminate execution by initiating an exit unwind.

Invalid Thread Stack

If, during the search for and invocation of frame-based handlers, the exception dispatcher detects that the thread's stack is corrupt, then the following steps take place:

1. The ESTAB_CONTEXT argument that is passed to all handlers subsequently invoked is set to 0.
2. The search for handlers immediately proceeds to the last-chance handlers

Handler Invocation and Arguments

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

Arguments:

EXPT_SIGNAL	This is a duplicate of the information passed in the EXPT.KIND field of the primary exception record.
EXPT_REC_ADDR	The address of a primary exception record.
ACTIVE_CONTEXT	A pointer to an invocation context block containing the saved original context at the point where the exception occurred.
ESTAB_CONTEXT	For frame-based handlers this is a pointer to an invocation context block containing the saved context of the routine where the handler was established. For vectored handlers, this argument is an indicator of stack validity. It is 0 if the system has detected that the thread's stack is corrupt during the search for frame-based handlers, and 1 if the stack is valid. Only vectored handlers are invoked if stack corruption has been detected.
ESTAB_PROC_VALUE	For frame-based handlers this is the procedure value for the routine where the handler was established. For vectored handlers, the contents of this argument is *unpredictable*.
HANDLER_DATA	For frame-based handlers, this is the address of the handler data in the establisher's procedure descriptor (the address of either PDSC.STACK_HANDLER_DATA or PDSC.REG_HANDLER_DATA). For vectored handlers, this is the data value specified when the handler was established.

Function Value:

STATUS	A status value indicating the action to be taken upon handler return. The valid status values are STATUS_CONTINUE and STATUS_RERAISE.

6.1.9 Modification of Exception Records and Context by Handlers

The exception records, exception qualifiers, and invocation context blocks that are passed to an exception handler are always allocated in writeable memory. Handlers may 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 defined below) but do not affect the original data structures.

The effect of a handler modifying passed exception information is as follows:

1. If EXPT.FLAGS.IS_NONRESUMABLE in the primary exception record is changed from 0 to 1, then the exception handler which made the modification *must not* return STATUS_CONTINUE, nor may any handler subsequently invoked for the exception return STATUS_CONTINUE.

   If STATUS_CONTINUE is returned after EXPT.FLAGS.IS_NONRESUMABLE has been changed from 0 to 1, then a nested exception is raised with EXPT.VALUE = STATUS_NONCONTINUE, indicating that an attempt was made to continue from a noncontinuable exception. This second exception is also noncontinuable.

2. If any flags in EXPT.FLAGS in the primary exception record are modified except as specified above, there is no effect after the exception handler completes; all handlers subsequently invoked for the exception receive a primary exception record with the flags unmodified.

   In particular, if an exception handler changes EXPT.FLAGS.IS_NONRESUMABLE from 1 to 0, that handler modification must not return STATUS_CONTINUE, and any and all handlers subsequently invoked for the exception will be invoked with EXPT.FLAGS.IS_NONRESUMABLE set to 1.

3. If the contents of the invocation context block specified by CONTEXT is modified by a handler, the results are *unpredictable*, and such a handler does not conform to this standard.

4. Except as specified above, 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 standard.

6.1.10 Handler Completion and Return Value

When an exception handler has finished all its processing, it must complete its execution by:

- *reraising* the exception,

  or

- *continuing* execution of the thread,

  or

- initiating procedure invocation *unwinding*.

Completion by Reraise

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

Reraise causes the next exception handler to be invoked (see Section 6.1.8, Invocation of Exception Handlers).

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

Completion by Continue

By returning STATUS_CONTINUE, an exception handler can continue execution of the thread at the address specified by EXPT.EPC in the primary exception record with the context of the interrupted procedure restored.

If STATUS_CONTINUE is returned and EXPT.FLAGS.IS_NONRESUMABLE is 1, then a nested exception is raised with EXPT.VALUE = STATUS_NONCONTINUE, indicating that an attempt was made to continue from a noncontinuable exception. This second exception is also noncontinuable.

Restrictions on Continuation

Continuation from Unwind

STATUS_CONTINUE does not apply when EXPT.KIND = EXC_DEC_UNWIND. When an unwind is in progress, the status returned by handlers is ignored and unwinding always proceeds as though STATUS_RERAISE was returned. That is, handlers may not continue during an unwind operation.

Continuation from Signal Exceptions

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

Completion by Unwinding

The Unwind type of completion is more complex than simply returning a value. See Section 6.2, Unwinding, for details and considerations on Unwinding.

6.1.11 Other Considerations

Exception Synchronization

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

Because of this, the state of the machine when a hardware exception occurs cannot be assumed with the same precision as it can be assumed on conventional VAX unless such precision has been guaranteed by bounding the exception range with the appropriate insertion of DRAINT 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 then it must issue a DRAINT instruction before it establishes itself as the *current procedure*.

  RATIONALE:

  The above 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 not contained directly within that procedure or by any procedure that might have been called while that procedure was *current* then it must issue a DRAINT instruction in the procedure epilogue while it is still the *current procedure*.

  RATIONALE:

  The above 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 then it must issue a DRAINT instruction immediately before and after any call. Furthermore, the handler must be able to recognize exception PC values that represent DRAINT 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.

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 which is not sensitive to invocation depth, an exception detected while that called procedure is current may have been caused by an instruction issued while the caller was the current procedure. This means that 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.

```
\\
It should be noted that many of the continuation cases that existed on
VAX implementations will no longer be needed on ALPHA as instruction
qualifiers can be used to force the traps never to occur.
\\
```

Continuation from Exceptions

The ALPHA architecture neither guarantees that instructions are completed in the same order in which they were fetched from memory nor that instruction execution is strictly sequential. Continuation after some exceptions is possible, but there are restrictions as reflected in the following discussions.

Software raised general exceptions are, by definition, synchronous with the instruction stream and can have a well defined continuation point. Thus, a handler may have the option of requesting continuation from a software raised exception. However, since compiler-generated code typically relies on error free execution of previously executed code, continuing from a software raised exception may produce unpredictable results and unreliable behavior unless the handler has explicitly fixed the cause of the exception in such a way as to be transparent to subsequent code.

Hardware faults on ALPHA follow the same rules as the strict interpretation of the conventional VAX rules. Loosely paraphrased, these rules state that if the offending exception is fixed, re-execution of the instruction (as determined from the supplied PC) will yield correct results. This does *not* imply that no instructions following the faulting instruction have been executed (see the *Digital Equipment Corporation ALPHA System Reference Manual* for more 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 since all the information required for a restart is not available. The most straightforward and reliable way in which software may guarantee the ability to continue from this type of exception is by placing appropriate DRAINT instructions in the code stream. Although this does allows continuation, this technique must be used with extreme caution due to the negative side effect on application performance.

6.1.12 Exception Handling Coexistence

The procedure based exception handling facility defined by this standard can coexist and interoperate with a global POSIX-style Signal Facility.

This section defines the features, and the limits, of such a coexistence.

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

LIB_SIG_TO_EXPT(..system_defined..)

This special signal handler will gather the software and/or hardware information associated with a signal and then pass it to the exception handling support code in such a way that it can be processed by normal exception handlers. The exception handling code will then proceed with its normal search and invocation procedures as described in Section 6.1.2, Exception Handling Overview.

If any handler returns with a *continue* indication (see Section 6.1.2), thread execution will resume 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 catchall handler is invoked, with *unpredictable* (system dependent) results.

- When an application which utilizes procedure based exception handling is initialized in an environment where signals are also supported, the run time system will install the special signal handler *LIB_SIG_TO_EXPT()* for each of the signals *SIGFPE*, *SIGSEGV*, *SIGBUS*, *SIGILL*, *SIGEMT*, *SIGIOT*, *SIGSYS*, and *SIGTRAP*.

  Applications and language run time support code may install *LIB_SIG_TO_EXPT()* for other signals if desired.

- If, for any signals for which *LIB_SIG_TO_EXPT()* is installed, a thread directly calls:

  signal(),
  sigblock(),
  sigsetmask(),
  sigpause(),
  sigvec(),

  or directly utilizes the Signal Facility support code in any way not otherwise described by this section, the Signal Facility mechanisms will continue to operate correctly but there will be *undefined* effects on procedure based exception handling of those signals.

  ```
  \\
  Possible effects include disabling of procedure based handling for the affected
  signal(s), by virtue of having modified the action taken for the signal(s).

  It is also possible, in a high-reliability environment, for the Signal Facility
  to be enhanced to permit a higher degree of interoperation with procedure based
  exception handling.  For example, an enhanced signal() routine might record the
  action specified in the direct call to signal(), and take that action if no
  frame-based handlers are located or if all frame-based handlers reraise the
  exception.

  However, this standard does not define the effects of these
  actions by a thread. These effects are system defined and
  therefore undefined for purposes of this standard.
  \\
  ```

- A hardware exception or asynchronous software interrupt may immediately pass control to the Signal Facility, in which case normal exception handlers will not be invoked unless *LIB_SIG_TO_EXPT()* is enabled as a signal handling procedure for the signal being delivered.

  ```
  \\
  This is how ULTRIX is likely to work.
  \\
  ```

Alternatively, the exception dispatcher may immediately initiate the search for exception handlers, and only pass control to the Signal Facility if all handlers re-raise the exception (i.e. do not continue execution of the thread).

```
\\
This is the current design of VIP.
\\
```

Which of these alternatives applies to a given environment is *system defined*, and handlers should not assume one or the other of these alternatives.

6.2 Unwinding

6.2.1 Unwind Requirements

The unwinding capabilities specified in this section are for support of:

- Correct and predictable nonlocal GOTO support in a multilanguage environment
- Support for the construction of modular, maintainable multilanguage applications

6.2.2 Unwind Basic Considerations

Unwinding refers to the action of returning from a procedure or a chain of procedures by a mechanism other than the normal return path. This concept is most often referred to in the context of exception handling and in fact is only allowed in the context of an active exception on some operating systems.

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 also results in the termination of all procedure invocations, including the invocation in which the unwind request was initiated, up to the target procedure invocation. Thread execution then continues at the target location.

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

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 is *undefined* by this standard.

In the ALPHA-64 flavor of the calling standard, unwinding does not require a exception handler to be active; it may be used by languages to implement nonlocal GOTO.

6.2.3 Types of Unwind

There are two types of unwind requests:

General unwind

A *general unwind* transfers control to a specified location in a specified procedure invocation.

The target procedure invocation is specified by an invocation handle.

The target location is specified with an absolute PC value.

When general unwind is completed, register R0 is restored from EXPT.VALUE in the primary exception record, allowing a status to be returned to the target of the unwind.

Exit Unwind

Since processes may be multithreaded, it is necessary for a thread which is terminating execution to clean up its use of shared resources.

Because of this, user mode thread exit may be accomplished *only* by unwinding. A special form of unwind, termed *exit unwind*, invokes all established frame-based handlers with an exception record specifying that an exit unwind is in progress, terminates all procedure invocations up to the beginning of the call chain, and terminates execution of the thread. Threads that use any other mechanism are not considered to be standard and their behavior is *undefined*.

Exit unwind terminates every procedure invocation in the invocation chain, after which execution of the thread is terminated.

6.2.4 Unwinds Invocation Types

There are two cases under which an unwind may be invoked. Those initiated while an exception is active and those initiated while no exception is active.

Unwind with No Active Exception

An unwind which is initiated when no exception is active is usually done to perform a non-local GOTO. i.e. to transfer control directly to some code location which is not part of the currently executing procedure. Even this type of operation must provide a mechanism to allow cleanup operations of terminated invocations (including restoring a consistent set of register values) to be performed. The unwind mechanism is used to support this type of operation.

Unwind during an Active Exception

The handler, or any descendant procedure called directly or indirectly by the handler, can continue execution of the thread at a different location than that at which the exception was raised by initiating an unwind operation.

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 which was established by any procedure invocations being terminated. These handlers are invoked with an exception record indicating that an unwind is in progress. This allows each procedure invocation being terminated to perform clean-up processing before its context is lost.

Once this phase has completed, the target invocation's saved context is restored and the execution is continued at the specified location.

One effect of the unwind operation is to terminate the exception handler invocation. Control is never returned to the point at which the unwind was initiated, and that handler invocation can therefore never return to its caller.

6.2.5 Unwind Initiation

Initiating a General Unwind

A thread may initiate a general unwind operation by calling a system library routine. This routine is defined as follows:

LIB_UNWIND(TARGET_INVO, TARGET_PC, EXPT_REC_ADDR)

Arguments:

TARGET_INVO	If non-zero, this is the invocation handle of the target procedure invocation to which the unwind should be done. If zero, an exit unwind is invoked.
TARGET_PC	If non-zero, the address within the target invocation at which to continue execution. If zero, the unwind is done to the return point of the active call in the target procedure. If TARGET_INVO is zero, then TARGET_PC is ignored.
EXPT_REC_ADDR	If non-zero, the address of a primary exception record. If zero, specifies that a default exception record should be supplied (see below).

Function Value:

None.

If the EXPT_REC_ADDR argument is zero, then *LIB_UNWIND()* supplies a default exception record which specifies exactly one exception record in which EXPT.VALUE = STATUS_UNWINDING.

When initiating an unwind operation, EXPT.EPC and EXPT.KIND are ignored in the specified primary exception record. EXPT.EPC is set to the return address of the call to *LIB_UNWIND()*. EXPT.KIND is set to EXC_DEC_UNWIND.

If the EXPT_REC_ADDR argument is specified when the unwind is initiated, then all other properties of the exception record are determined by EXPT_REC_ADDR. If *LIB_UNWIND()* detects that a specified exception record is not a valid unwind record, it will raise the a exception of STATUS_INVALID_EXCEPTION.

Once an unwind is initiated, control never returns to the point at which the unwind was initiated.

Initiating an Exit Unwind

A thread may initiate an exit unwind operation by calling a system library function defined as follows:

LIB_EXIT_UNWIND(STATUS)

Arguments:

STATUS	If non-zero, an unwind status value. If zero, indicates that the default unwind status value should be used.

Function Value:

None.	This function will never return.

The exit unwind operation terminates the thread after all active procedures have been terminated.

As the procedures in a thread are terminated, they must respond to the exit unwind by performing the appropriate exit actions.

Multiply Active Unwind Operations

During an unwind operation, another unwind operation may be initiated. This may occur, for example, if a handler which is invoked for the original unwind initiates another unwind, or if an exception is raised in the context of such a handler and a handler invoked for that exception initiates another unwind operation.

An unwind which is initiated while a previous unwind is active is either a *nested unwind* or an *overlapped unwind*.

Nested Unwind

A nested unwind is a general unwind which is initiated while a previous unwind is active, and whose target invocation in the procedure invocation chain is not a predecessor of the most current active unwind handler. That is, a nested unwind is one which does not terminate any procedure invocation which would have been terminated by the previously active unwind.

When a nested unwind is initiated, no special rules apply. The nested unwind operation proceeds as a normal unwind operation, and when execution resumes at the target location of the nested unwind, the nested unwind is complete and the previous unwind is once again the most current unwind operation.

Overlapping Unwind

An overlapping unwind is an exit unwind or general unwind which is initiated while a previous unwind is active, and whose target invocation in the procedure invocation chain is a predecessor of the most current active unwind handler. That is, an overlapping unwind is one which terminates one or more procedure invocations that would have been terminated by the previously active unwind.

An overlapping unwind is detected when the most current active unwind handler is terminated. This detection of an overlapping unwind is termed a *collision*.

When a collision occurs, the two unwind operations are merged into a new unwind operation, which is the only unwind operation active following the merge.

Collision with General Unwind

When an unwind collides with an already active general unwind, the target invocation of the merged unwind operation is whichever of the target invocations specified by the colliding unwind operations is the oldest predecessor on the procedure invocation chain. That is, the merged target invocation is whichever of the two target invocations causes the greatest number of invocations to be terminated.

(Note that exit unwind, by definition, always specifies an older predecessor than any general unwind).

The target location of the merged unwind operation is the target location associated with the oldest predecessor target invocation. If the colliding unwinds specify the same target invocation, then the target location in that invocation is the target location specified by the overlapping unwind, and the target location specified by the previously active unwind is discarded.

The EXPT_REC_ADDR argument of the merged unwind operation is that associated with the overlapping unwind. The argument associated with the previously active unwind is discarded.

After the colliding unwinds are merged, the unwind operation continues from the point of the collision.

In POSIX-conforming environments, if any unwind overlaps a general unwind for which EXPT.FLAGS.IS_LONGJMP is 1, the handler which initiated the colliding unwind is in error and *does not conform* to this standard.

Collision with Exit Unwind

When a general unwind collides with an already active exit unwind, a noncontinuable STATUS_COLLIDED_EXIT_UNWIND exception is raised at the point of the collision.

If an exit unwind collides with an already active exit unwind, the EXPT_REC_ADDR argument of the merged unwind operation is that associated with the overlapping unwind. The arguments associated with the previously active exit unwind are discarded.

6.2.6 Handler Invocation During an Unwind

When an unwind operation takes place, all frame-based exception handlers are invoked which were established by any procedure invocation being terminated. These handlers are invoked in the reverse order from which they were established.

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

Steps for Locating and Invoking Handlers for Unwind

When an unwind operation it initiated, the steps that implement the above capabilities are as follows:

1. If the EXPT_REC_ADDR argument is zero, supply a default exception record with EXPT.VALUE = STATUS_UNWINDING.
2. Let EXPT.KIND = EXC_DEC_UNWIND.
3. Let EXPT.EPC = the return address from the call to *LIB_UNWIND()*.

4. Let *current_invocation* be the procedure invocation which initiated the unwind operation.
5. If *current_invocation* = TARGET_INVO and EXPT.FLAGS.IS_EXIT_UNWIND is 0, go to step 20.
6. If *current_invocation* establishes a handler, invoke that handler.
7. If *current_invocation* is an active unwind handler, go to step 12.
8. Remove *current_invocation* from the procedure invocation chain.
9. If the procedure invocation chain is empty, go to step 24.
10. Let *current_invocation* be the most current invocation in the procedure invocation chain.
11. Go to step 5.
12. If EXPT.FLAGS.IS_EXIT_UNWIND is 1, go to step 8.
13. Let *old_expt_rec* be the EXPT_REC_ADDR argument passed to *current_invocation*.
14. Let *old_invo* be the TARGET_INVO argument passed to *current_invocation*.
15. Let *old_pc* be the TARGET_PC argument passed to *current_invocation*.
16. If *old_expt_rec*.FLAGS.IS_EXIT_UNWIND is 0, go to step 18.
17. Raise noncontinuable STATUS_COLLIDED_EXIT_UNWIND.
18. If *old_invo* denotes a predecessor of TARGET_INVO, let TARGET_INVO = *old_invo*, and let TARGET_PC = *old_pc*.
19. Go to step 8.
20. Restore saved context of *current_invocation*.
21. Set register R0 to the contents of EXPT.VALUE.
22. If TARGET_PC specified $\neq$ 0, let *return_pc* = TARGET_PC, otherwise let *return_pc* = saved return address for *current_invocation*.
23. Exit these steps, resuming execution at *return_pc*.
24. If EXPT.FLAGS.IS_EXIT_UNWIND is 0, raise STATUS_FRAME_NOT_FOUND exception.
25. Terminate execution of thread.

Invalid Thread Stack

If, during the search for and invocation of frame-based unwind handlers, the system detects that the thread's stack is corrupt, the following steps take place:

1. The STATUS_STACK_INVALID exception is raised, interrupting the unwind operation.
2. The ESTAB_CONTEXT argument that is passed to all handlers subsequently invoked is set to 0.
3. The search for handlers immediately proceeds to the last-chance handlers.

6.2.7 Unwind Completion

When an unwind completes the following conditions are true:

- The target procedure invocation is the most current invocation in the procedure invocation chain.
- The environment of the target invocation is restored to the state when that invocation was last current, except for the contents of all scratch registers
- R0 contains a value which was passed (either directly or indirectly) by the routine that invoked the unwind.
- Execution continues at the target location

6.2.8 Unwinding Coexistence with Setjmp/Longjmp

The procedure invocation unwinding facility defined by this standard can coexist and interoperate with *setjmp / longjmp* facilities.

The features and limits of this coexistence are defined as follows:

- A handler which modifies the contents of any exception record associated with a *longjmp* unwind does not conform to this standard.
- Any environment which conforms to this standard must implement the *longjmp()* function by initiating a *longjmp unwind*. This means that *longjmp()* must construct an exception record with EXPT.FLAGS.IS_LONGJMP set to 1 and call *LIB_UNWIND()* with the appropriate target. All handlers established by procedure invocations being terminated will be invoked.

  The unwind facility supports unwinding operations that maintain the semantics of *longjmp()*, in order to make this interoperation possible.

6.2.9 Compatibility with Other Environments

The exception handling and unwinding facilities defined by this standard are designed to allow emulation of older exception handling and unwinding facilities. Examples of how such emulation can be accomplished are presented below.

Compatibility with VAX/VMS

An exception handling and unwinding environment which provides the interfaces of the VAX/VMS Condition Handling Facility can be layered on the facilities defined by this standard.

Routines to emulate *SYS$UNWIND()* and other VAX/VMS exception handling routines can be provided using the information and facilities specified.

Dynamic exception information conversion can be done for any frame by establishing a static handler which converts the static handler's arguments to a VAX/VMS signal array and mechanism array, and then invokes a dynamically-established VAX/VMS exception handler. This VAX/VMS handler can be dynamically established and revoked, for example, by use of some location in the establisher's stack frame; the offset of that location in the stack frame can be specified via the HANDLER_DATA argument passed to the static handler. Implementors should note that for a program to conform to this standard implies that anytime a dynamic handler of this type is established or revoked a DRAINT instruction must be executed.

Compatibility with ULTRIX/RISC Exception Handling

An exception handling and unwinding environment which provides the interfaces of ULTRIX /RISC Exception Handling can be layered on the facilities defined by this standard.

Routines to emulate *exc_resume()* and other ULTRIX/RISC exception handling routines can be provided using the information and facilities specified.

Dynamic exception information conversion can be done for any frame by establishing a static handler which converts the static handler's arguments to the arguments expected by ULTRIX exception handlers, and then invokes an ULTRIX/RISC exception handler. The DATA argument to the static handler can specify the location of an ULTRIX/RISC exception handler information array, which the static handler can process using the same logic as the ULTRIX/RISC *exception_dispatcher()* routine.

6.3 Asynchronous Software Interrupts

Asynchronous software interrupts can arise from a number of system and environment specific mechanisms. Examples of mechanisms which may cause such interrupts are:

- Usage of a POSIX style signal facility
- Usage of a VMS style AST mechanism

Because many of the usage semantics are the same, a given execution environment may chose to provide a connection mechanism between asynchronous software interrupts and the exception handling facility. Such a coexistence is described in Section 6.1.12, Exception Handling Coexistence.

While the existence of asynchronous software interrupts profoundly effects this standard, conventions for their usage, invocation and handling other than mentioned above are beyond the scope of this document.

CHAPTER 7

MULTITHREADED ENVIRONMENT CONVENTIONS

```
\\
This is a capability that does not exist yet on either VMS or ULTRIX.

Language bindings for library routines, status values, etc. still need to be
agreed upon. The naming conventions used in this version of this standard are
formal, for purposes of specification only, and do not represent the final
language bindings to these interfaces.
\\
```

This standard defines the essential mechanisms and conventions to allow support of the coordination of the execution of multiple threads in a multilanguage environment, including:

- Access to the thread private environment
- Thread stack limit checking
- Thread handling of asynchronous interrupts
- Thread exit
- Management of thread private storage
- Implementation of the Concert Multithread Architecture Services

  Only the essential mechanisms to support implementation of the *Digital Equipment Corporation Concert Multithread Architecture Services* as a layered set of services are defined by this standard.

7.1 Thread Environment Block

The *thread environment block* (TEB) is a data structure associated with each thread. A TEB is provided to each thread by the system, is used to maintain user mode software context specific to that thread, and to manage resources used by the thread.

7.1.1 Access to the Thread Environment Block

A thread can obtain the address of its own TEB by calling an operating system supplied function defined as follows:

LIB_GET_TEB()

Arguments:

None.

Function Value:

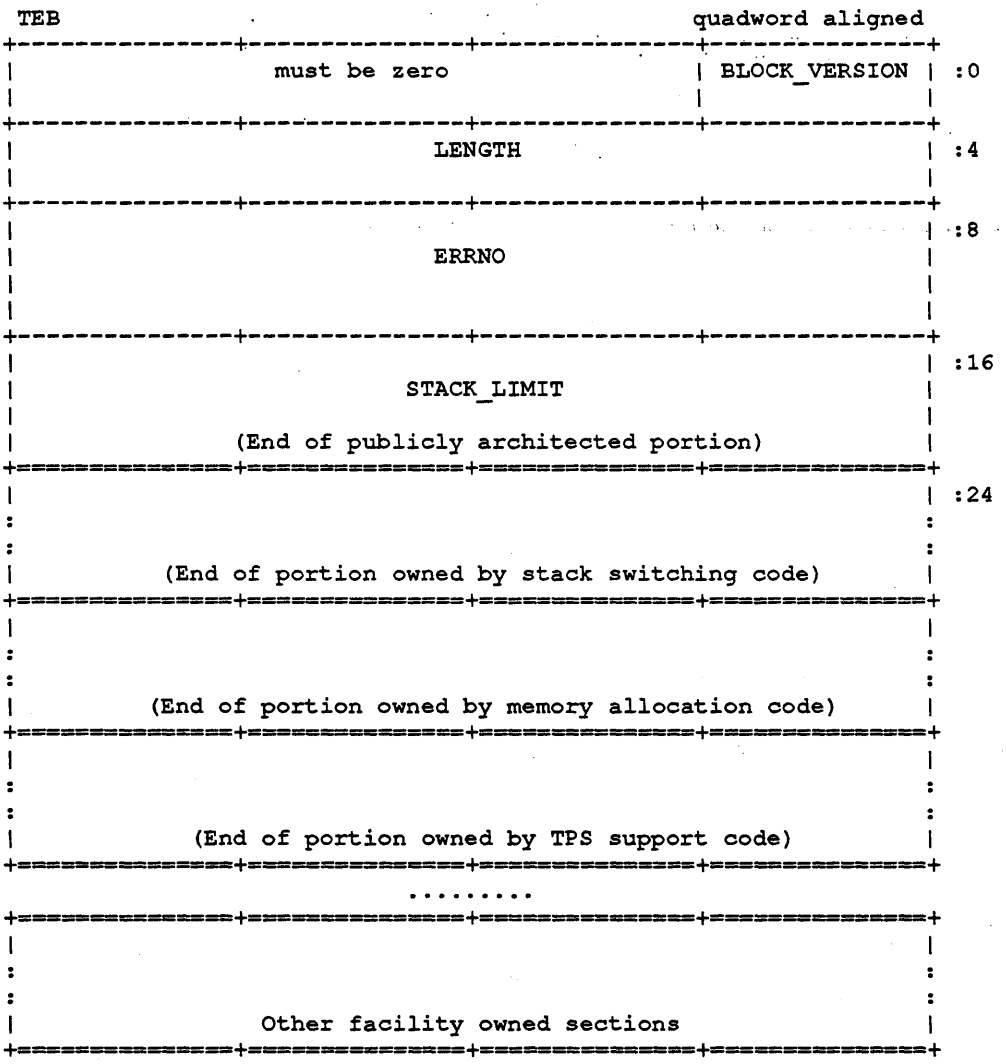
TEB	The address of the Thread Environment Block.

The TEB is fixed at the same virtual address for the life of the thread, so the address returned by *LIB_GET_TEB()* is valid until the thread terminates.

7.1.2 Thread Environment Block Format

The following figure illustrates the layout of the thread environment block:

Figure 7–1: Thread Environment Block Format



TEB.BLOCK_VERSION is a byte that indicates the TEB version. This value would change if the format of this structure ever changes. For now, it is always set to the value 1.

TEB.LENGTH contains the length in bytes of the thread environment block. TEB.LENGTH is system defined, is the length allocated for the block (not necessarily the length of the fields currently defined in the block), and is subject to change if the TEB is extended.

TEB.ERRNO contains the current value of the global *errno* variable for this thread.

The contents of TEB.ERRNO are *unpredictable* except in POSIX-conforming environments.

TEB.STACK_LIMIT contains the lowest address allocated to the stack that is currently active.

7.2 Stack Limit Checking

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

Detection of a stack overflow situation is important. Without it, a thread, writing into what it considered to be stack storage, could modify data allocated in that memory for some other purpose. This would most likely produce unpredictable and undesirable results and/or unreproducable application failures.

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

7.2.1 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*.

7.2.2 Methods for Stack Limit Checking

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

There are two methods for stack limit checking, depending on the circumstances of the stack extension.

Implicit Stack Limit Checking

If the stack is being extended by an amount known at compile time to be less than or equal to MAX_NOCHK_EXTEND, then no explicit stack limit checking is required. Stack limit checking will be implicit, based on the fact that any reference to the guard region will cause a hardware exception.

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

These conventions are defined as follows:

1. Explicit stack limit checking must be done unless the amount by which the SP is decremented is known to be less than or equal to MAX_NOCHK_EXTEND.
2. The lowest addressed byte in the *new stack region* (the region of the stack from the old value of SP – 1 to the new value of SP) must be accessed before SP can be decremented for a subsequent stack extension.

   This access can be done either before or after the SP is decremented for this stack extension, but it must be done before it can be decremented again.
3. No standard procedure call can be done before the lowest addressed byte in the new stack region is accessed.

4. The system exception dispatcher ensures that the lowest addressed byte in the new stack region is accessed if any kind of asynchronous interrupt occurs after SP is decremented but before the access to the new lowest byte occurs.

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

If the stack is being extended by an amount known at compile time, but known to be greater than MAX_NOCHK_EXTEND, then a code sequence which follows these conventions can be executed in a loop to safely extend the stack incrementally in segments smaller than or equal to MAX_NOCHK_EXTEND.

The stack must **not** be extended incrementally in procedure prologues. A procedure prologue that needs to extend the stack by an amount greater than MAX_NOCHK_EXTEND may test new stack regions in a loop that does not modify SP, and then extended the stack with one instruction.

As a matter of practice, the system can provide multiple guard pages in the guard region. When a stack overflow is detected as a result of access to the guard region, one or more guard pages can be unprotected for use by the exception handling facility, and one or more guard pages can remain protected to provide implicit stack limit checking during exception processing. However, the size of the guard region and the number of guard pages is *system defined*, and is not defined by this standard.

Explicit Stack Limit Checking

If the stack is being extended by an amount of unknown size or known size greater than MAX_NOCHK_EXTEND then the following steps must be performed in this order:

1. The new SP value must be calculated and checked against the actual stack limit, which is available in TEB.STACK_LIMIT in the thread environment block.
2. If the new value for SP is an address lower than the stack limit, then the STACK_OVERFLOW exception must be raised which will initiate orderly thread termination.
3. Otherwise, the stack can be extended by setting SP to the value calculated in step 1.

7.2.3 Stack Overflow Handling

If a stack overflow occurs, the result is *system defined*.

The system may transparently handle the overflow, extending the thread's stack, resetting TEB.STACK_LIMIT appropriately, and continuing execution of the thread.

Note that, if this action is taken, a stack overflow that occurs in a called procedure might cause the stack to be extended and TEB.STACK_LIMIT to be modified in the TEB.

Because of this, TEB.STACK_LIMIT must be considered volatile and potentially modified by external procedure calls and by handling of exceptions.

7.3 Asynchronous Software Interrupts

Asynchronous Software Interrupt usage, handling, and other interactions with the multithread environment is outside the scope of this document. These interactions and conventions are described in the implementation documentation for the *Digital Equipment Corporation Concert Multithread Architecture Services*

7.4 Thread Exit

When a thread exits, various actions must be performed. Such actions may include:

- Terminate any threads created as computational resources specifically for the thread being terminated
- Free all thread private storage regions
- Release locks and mutexes acquired by the thread
- Free other resources local to the thread

In order to ensure that all procedures which are active when a thread exits are able to perform the necessary exit actions, the *only* defined means for performing thread exit is to initiate an exit unwind operation, as described in Section 6.2, Unwinding.

Systems which provide alternative, system-specific services for thread termination must implement such services by initiating an exit unwind.

7.5 Thread Private Storage

Thread private storage (TPS) provides threads in a multithreaded environment with per-thread storage that has the properties of static storage. That is, each thread has its own copy of all thread private storage, which is shared by the procedures that run in the context of that thread.

Since TPS is thread private, changes to such storage made by a procedure running in the context of one thread are independent of changes made by the same procedure running in the context of a different thread.

TPS is composed of *TPS regions*, where each region has many of the properties of a program static section such as:

- Each TPS region has a global name. Any number of modules can contribute to a TPS region by specifying the same global name.
- Contributions to the same TPS region from different modules can be overlaid or concatenated.
- The total size of a region is determined by the linker.
- A TPS region can be aligned as required by the contributing modules. The alignment of the region is determined by the linker.
- The initial contents of a TPS region can be specified. Each thread's copy of a TPS region (termed an *instance*) is initialized to these contents.
- Access to a TPS region is always *read/write*.
- The offsets for data within a region are managed by compilers and the linker using the general methods available for managing data within program sections (such as variables within a COMMON block).
- Dynamically loaded shared libraries can define TPS regions. These TPS regions are disjoint from the TPS regions of the calling program and of other dynamically loaded shared libraries. That is, independently linked libraries do not contribute to each other's TPS regions.

TPS regions have the following additional properties not shared by program static storage sections:

- Thread private storage must be allocated and initialized before being accessed by a thread.
- Each thread accessing a TPS region accesses a distinct instance of the region.
- References in object code to a TPS region must be made using special relocations supported by the linker.
- References to data within a TPS region must be made as offsets relative to a base pointer. The base pointer is established when the region is allocated, and must be obtained by each procedure that accesses the region.
- Each TPS region allocated by a thread will be freed when the thread terminates.

TPS regions are not typically visible to the high level language user. This visibility is consistent with the visibility of program sections to the high level language user. High level languages are expected to use TPS regions when the semantics of the language require thread private static storage. However, attributes or directives may be provided as language extensions to make thread private static storage visible to the user.

7.5.1 Referencing Thread Private Storage

Accessing Data in TPS Regions

Resolving a TPS location requires the base address of the TPS region which contains the location, and the offset of the data in the TPS region. The address of the TPS region is used as a base pointer for all access to data in the region.

Referencing thread private storage requires special techniques because, unlike other memory references, a reference to TPS may require creation of a TPS region.

Referencing TPS Regions

A reference to a TPS region actually resolves to the address of a *TPS descriptor* for that region. A system library function may be used to determine the base address of the TPS region associated with a TPS descriptor (the structure of which is private to the system and is not defined by this standard).

This system supplied function is defined as follows:

LIB_GET_TPS(TEB, TPS_DESCRIPTOR)

Arguments:

TEB	The address of the thread environment block.
TPS_DESCRIPTOR	The address of a TPS descriptor.

Function Value:

TPS_REGION_ADDRESS	The base address of the specified TPS region.

If the specified TPS region does not already exist in the calling thread, *LIB_GET_TPS()* will create and initialize the region.

```
\\
There are a number of techniques which may be used to minimize the number of
calls needed to access TPS regions. See Section D.1.5, Optimized TPS Access, for
details.
\\
```

7.5.2 Sharing TPS Regions Between Shared Libraries

Sharing TPS regions across shared libraries only resolves TPS regions between programs or libraries that are explicitly linked together. Two or more libraries that are never linked together, but are dynamically loaded into the same address space, do not share TPS regions. This behavior is consistent with the behavior of FORTRAN COMMON.

In addition, if a shared library containing a TPS region is relinked, but a program linked against it is not, the program behavior is *undefined*, just as it is for FORTRAN COMMON in the same circumstances. The developer of the shared library must control this situation by the use of shared library match control.

7.5.3 Deleting TPS Regions

TPS region instances have lifetimes no longer than that of the thread to which they are bound. They are deleted (by freeing the thread private virtual memory zone) when the thread terminates.

Deleting TPS regions upon thread exit is the responsibility of the run time facility that supports TPS.

CHAPTER 8

RUN TIME GENERATED CODE

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

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

Code that is generated at run time and stored into writeable memory must be made safe to use before it can be executed. This is because the system does not necessarily maintain coherency between writeable memory and the instruction cache; it assumes, for example, that replacing a writeable virtual page with a different virtual page in the same physical page does not require instruction cache invalidation. It is the responsibility of an application that generates run time code to obey the correct protocols to insure that the newly constructed code may be executed properly. Failure to do this can result in application failure.

The system library functions described below provide services to ensure that applications can declare new code to the system so that call chain navigation can be performed correctly. These routines do not address the various techniques required for cache coherency, etc.

8.1 Procedure Descriptors for Run Time Generated Code

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

Before run time generated code may be executed, a system library function must be called one or more times to associate each valid PC range with a procedure descriptor. This can be done by calling the system supplied procedure defined as follows:

LIB_ADD_PC_RANGE(PROC_DESC_ADDR, START, END)

Arguments:

PROC_DESC_ADDR	The address of a procedure descriptor that describes to code in the PC range.
START	The lowest address in the PC range
END	The highest address in the PC range

Function Value:

STATUS	A status value: *STATUS_SUCCESS* indicates that the PC range has been properly added to the PC mapping tables. *STATUS_FAILURE* indicates that the requested operation failed.

When a PC range added by *LIB_ADD_PC_RANGE()* no longer contains valid code or the code will not be executed again, a system library function should be called to remove the PC mapping information. This routine is defined as follows:

LIB_REMOVE_PC_RANGES(START, END)

Arguments:

START	The low address of the PC ranges to invalidate
END	The high address of the PC ranges to invalidate

Function Value:

STATUS	A status value: *STATUS_SUCCESS* indicates that the range has been removed, or that none existed for the values given. *STATUS_FAILURE* indicates that the requested operation could not be performed.

This procedure may be used to removed:

- All dynamically added PC ranges, by using START = 0, and END = -1.)
- All dynamically added PC ranges within a given section of memory, by using START = low address, and END = high address.)

- One dynamically added PC range, by using START = low address of range, and END = high address of range.)

A system library function may be called to release all dynamically added PC ranges associated with a given procedure descriptor. This function is defined as follows:

LIB_REMOVE_PDSC_PC_RANGES(PROC_DESC_ADDR)

Arguments:

PROC_DESC_ADDR	The address of the procedure descriptor for which all dynamically added PC ranges are to be removed.

Function Value:

STATUS	A status value: *STATUS_SUCCESS* indicates that all PC address ranges have been removed that were associated with the given procedure descriptor. *STATUS_FAILURE* indicates that the arguments were invalid.

8.2 Steps for Run Time Generation of code

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

1. Allocate memory for code
2. Write code and procedure descriptor(s) to memory
3. Call LIB_ADD_PC_RANGE( )
4. Repeat step 4 for each code segment
5. Execute code
6. Call LIB_REMOVE_PC_RANGES( )
7. Deallocate memory containing code

CHAPTER 9

CODE EXAMPLES

The following example illustrates many of the techniques and conventions mentioned above. This example does *not* presume to represent how a compiler might actually layout memory, or to represent actual ALPHA assembler notation. The code fragments presented do not presume to do anything useful.

```
MODULE X

;       Linkage section for Module X

        .psect      $LINK           ;Read-Only after image activate

X1::                                ;Descriptor for stack frame procedure X1

        .word   <<<PDSC_FLAGS_NATIVE | PDSC_FLAGS_NO_JACKET>@4> -
                        | PDSC_KIND_PC_STACK > ;flags + kind
        .word   8                   ;Register save area offset
        .long   26                  ;No signature info + Entry Reg

        .address_q X1_ENTRY         ;Address of entry code sequence for X1
        .long   48                  ;Size of fixed part of stack
        .word   0                   ;SP set
        .word   24                  ;Entry length
        .long   <BIT13|BIT14|BIT15|BIT29> ;Integer Register mask
        .long   0                   ;Floating Register mask

Y1_LKP:                             ;Linkage pair for procedure Y1
        .address_q Y1_ENTRY         ;Code address for optimized call
        .address_q Y1               ;Y1's descriptor address(procedure value)

POWER_2_TABLE:                      ;Read-only table of powers of 2
        .long 1,2,4,8,16,32,64,128,256,512,1024,2048,4096,8192

Y_DATA_1_ADDR:                      ;External static variables
        .address_l Y_DATA_1         ;Address of Y_DATA_1
Y_DATA_2_ADDR:
        .address_l Y_DATA_2         ;Address of Y_DATA_2


;       Entry code sequence for procedure X1

        .psect  $CODE               ;Code for Module X

X1_ENTRY::                          ;Address of X1's descriptor is in R27
                                    ; on standard call. Return address is in R26
        LDA     SP,-48(SP)          ;Allocate stack space for X1 stack frame
        STQ     R26,08(SP)          ;Save return address
        STQ     R13,16(SP)          ;Save preserved R13
        STQ     R14,24(SP)          ;Save preserved R14
        STQ     R15,32(SP)          ;Save preserved R15
                                    ;X1 is now the current procedure

        MOV     R27,R13             ;Move own procedure descriptor address to a
                                    ; preserved register because it is needed
                                    ; after the call

;       Calculate 2 raised to power of first argument and store in Y_DATA_1
```

```
        LDA     R23,POWER_2_TABLE-X1(R13) ;Scratch R23 now has address of
                                          ; powers of 2 table
        SLL     R16,#2,R24        ;Scratch R24 now has offset into table
        ADDL    R23,R24,R23       ;R23 now has address of answer
        LDL     R23,(R23)         ;Fetch answer from memory
        LDL     R24,Y_DATA_1_ADDR-X1(R13) ;R24 now has address of Y_DATA_1
        STL     R23,(R24)         ;Store answer in Y_DATA_1

;       Preserve arguments across call to Y1

        MOV     R16,R14           ;Save arg 1 in preserved R14
        MOV     R17,R15           ;Save arg 2 in preserved R15

;       CALL procedure Y1

        LDQ     R26,Y1_LKP-X1(R13) ;Pick up entry code address
        LDQ     R27,Y1_LKP-X1+8(R13) ;Load his procedure value
        CLR     R25               ;Load AI register
        JSR     R26,(R26)         ;Make call to Y1 leaving return address in R26

;       Store first plus second argument in external Y_DATA_2 and
;       return sum as X1 value

        ADDL    R14,R15,R0        ;Get sum in return value register
        LDL     R23,Y_DATA_2_ADDR-X1(R13)
                                  ;Load scratch R23 with address of Y_DATA_2
        STL     R0,(R23)          ;And store answer in Y_DATA_2 as well

        LDQ     R23,08(FP)        ;Get return address
        LDQ     R13,16(FP)        ;Restore preserved R13
        LDQ     R14,24(FP)        ;Restore preserved R14
        LDQ     R15,32(FP)        ;Restore preserved R15
        LDA     SP,48(SP)         ;Restore caller's SP
        RET     R31,(R23),signature_hint ;Return to caller with signature
                                  ; (Sum in R0)
```

MODULE Y

```
        .psect  $LINK             ;Linkage section for Module Y

Y1::                              ;Descriptor for register frame procedure Y1

        .word   <<<PDSC_FLAGS_BASE_REG_IS_FP | PDSC_FLAGS_NATIVE | -
                   PDSC_FLAGS_NO_JACKET | PDSC_FLAGS_HANDLER_VALID>@4> -
                   | PDSC_KIND_PC_REGISTER > ;flags + kind
        .byte   0                 ;MBZ
        .byte   26                ;Return address is left undisturbed in R26
        .long   26                ;No signature info + Entry Reg

        .address_q Y1_ENTRY       ;Address of entry code sequence for Y1
        .long   0                 ;Fixed frame Size of Zero
        .word   0                 ;SP set
        .word   4                 ;Entry length
        .quad   Y1_HND-.          ;Offset to Y1_HND exception handler

Y1_HND:                           ;Procedure descriptor for exception handler

        .word   <<<PDSC_FLAGS_BASE_REG_IS_FP | PDSC_FLAGS_NATIVE | -
                        PDSC_FLAGS_NO_JACKET>@4> -
                        | PDSC_KIND_PC_REGISTER > ;flags + kind
        .byte   0                 ;MBZ
        .byte   26                ;Leave return address undisturbed in R26
        .long   26                ;No signature info + Entry Reg
```

```
        .address_q Y1_HND_ENTRY ;Address of entry code sequence for Y1_HND.
                                ; (Y1_HND_ENTRY is not shown
                                ; in this example.)
        .long   0               ;Fixed frame Size of Zero
        .word   0               ;SP set
        .word   nn              ;Entry length

Y_STATIC_DATA:                  ;Address of base of static Y data
        .address_l Y_DATA_BASE

        .psect  $DATA           ;Static data area

Y_DATA_BASE:                    ;Start of static data area

Y_DATA_1::                      ;Global static variable Y_DATA_1
        .long   0               ; (filled in above)
Y_DATA_2::                      ;Global static variable Y_DATA_2
        .long   37


;       Sample code sequence for procedure Y1

        .psect  $CODE           ;Code for Module Y

Y1_ENTRY::                      ;Address of Y1's descriptor is in R27
                                ; on standard call. Return address is in R26
        DRAINT                  ;Force pending exceptions to be raised
                                ; in context of caller

;       Compute product of global static Y_DATA_1 and Y_DATA_2.
;       Store product in Y_DATA_1.

        LDL     R23,Y_STATIC_DATA-Y1(R27)
                                ;Scratch R23 now has base of data area
        LDL     R24,Y_DATA_1-Y_DATA_BASE(R23) ;Scratch R24 gets Y_DATA_1 value
        LDL     R28,Y_DATA_2-Y_DATA_BASE(R23) ;Scratch R28 gets Y_DATA_2 value
        MULL/V  R24,R28,R24     ;Calculate product
        STL     R24,Y_DATA_1-Y_DATA_BASE(R23) ;Store product in Y_DATA_1
        DRAINT                  ;Synchronize exceptions

;       Return to caller

        RET     R31,(R26),signature_hint ;Return to caller with signature
```

Part II: ALPHA-32 Calling Standard

This part of the document describes the ALPHA-32 Calling standard.

Wherever possible this entire section of the document only specifies changes or differences from existing specifications such as the VAX/VMS calling standard or the ALPHA-64 flavor of this calling standard. Further reading in related documents may be required to fully understand the information in this part.

First time readers are strongly encouraged to read Part I, ALPHA-64 Calling Standard, in its entirety before attempting to assimilate the material presented below.

CHAPTER 10

INTRODUCTION

The ALPHA-32 flavor of the calling standard is intended to provide a calling standard that provides a high degree of compatibility with the VAX/VMS environment. Conventions that have been changed are, for the most part, those that have been dictated by differences between the ALPHA and VAX hardware architectures.

Many of these conventions are visible to the high level language programmer, and therefore cannot be changed without requiring source level changes in high level language programs.

Since source level compatibility and portability between VAX and ALPHA are explicit goals, users should not depend on the properties of this architecture except indirectly through high level language facilities that are portable across architectures.

10.1 Applicability

It is believed that ALPHA/VMS phase 1 will be the only operating system to ever support this flavor of the calling standard.

CHAPTER 11

BACKGROUND

11.1 Goals

The calling standard for the ALPHA-32 environment has one dominating goal which is to provide compatibility with VAX/VMS. In order that this goal may be accomplished other related goals must also be met. Thus the goals for the ALPHA-32 flavor of the calling standard are:

1. Provide a 32-bit user mode environment providing a *very* high degree of compatibility with current VAX/VMS programs
2. Simplify coexistence with VAX procedures executing under the translator environment.
3. Simplify the translation of VAX assembler to native ALPHA object code.

After these goals have been met, this flavor of the calling standard attempts to provide the performance, portability, and other goals of the ALPHA-64 flavor of the calling standard (see Section 2.1) as long as they do not do not conflict with the goals stated above.

11.2 Constraints

This standard was developed under the following constraints:

As well as those listed under Section 2.2, Constraints, the following considerations were factored into the development of this flavor of the calling standard:

- There is a very large body of code that exists for current VAX/VMS systems. The investment represented by this code must not be wasted.
- This flavor of the calling standard MUST provide a high degree of compatibility to encourage a well orchestrated migration from conventional VAX/VMS systems to ALPHA /VMS systems.

11.3 Tradeoffs

While the overriding goal of the ALPHA-32 environment flavor of the calling standard is for source compatibility with VAX/VMS there are certain tradeoffs that have been made. These tradeoffs are documented below:

Table 11–1: ALPHA-32 Calling Standard Tradeoffs

Issue	Reason
No emulation of VAX frames	Exact emulation of VAX stack frames was considered for compatibility reasons. The decision was made not to do this emulation based on the following facts: to do so would be extremely costly in performance, the stack frame format is not visible when using high level languages, hardware differences would make the emulation technically very difficult, there seems to be no real requirement for this feature.
First n args in regs	Passing all call arguments in memory was considered for compatibility reasons. Passing some arguments in registers was deemed permissible for the following reasons: the location of arguments is not visible when using high level languages, a significant performance advantage can be realized by passing some arguments in registers, there is an acceptable fallback mechanism that may be utilized in those cases where it is required to access a totally in-memory argument list.
Return values in R0 and F0	VAX/VMS style (R0 + R1) return values were considered for compatibility reasons. That choice was rejected for the following reasons: location of return value is not visible when using high level languages, the ALPHA hardware provides no means of transferring data between floating point and integer registers without going through memory, ALPHA registers are 64 bits wide and thus there appears to be no strong motivation to split a single 64-bit return value between two registers.
New mechanism array format	One might consider trying to emulate the VAX/VMS mechanism array format but that would appear to be a very bad choice due to the facts that: the ALPHA Hardware has a different register set and register sizes, the calling standard is different than the VAX/VMS one, only a small amount of code is sensitive to the format of this data structure and that code will most likely have to change anyway due to hardware differences.
Argument list width is 64 bits	32-bit wide argument lists were considered for compatibility reasons. 64-bit wide argument lists were chosen because: Argument list width is not visible when using high level languages and is only visible to middle level (C) languages in very contorted ways. Not so distant future language feature additions, such as *INTEGER*8* support for FORTRAN, would make 32-bit wide arguments very cumbersome.
New function stack return mechanism	A new stack return mechanism was chosen because it allowed compilers to utilize techniques developed for the ALPHA-64 flavor of the calling standard as well as increasing the interoperability of mixed language calls.
New argument passing rules	New argument passing rules were chosen to maximize the speed advantage over conventional VAXes that the ALPHA architecture offers.

Table 11–1 (Cont.): ALPHA-32 Calling Standard Tradeoffs

Issue	Reason
New function value returning rules	New function value returning rules were chosen to maximize the speed advantage of ALPHA as well as maximizing the interoperability of mixed language procedures.

11.4 Important Technical Decisions

This section discusses important technical decisions that had dramatic effect on this flavor of the calling standard.

11.4.1 PC Map vs. FP Based Call Chain Navigation

Current plans indicate that the ALPHA-32 flavor of the calling standard will only be supported by the ALPHA/VMS operating system. In support of the strong goal for VAX/VMS compatibility wherever possible together with a severe time-to-market constraint it seemed clear that the ALPHA-32 flavor of the calling standard should use the FP based method.

CHAPTER 12

BASIC CONSIDERATIONS

This section describes some fundamental concepts of the ALPHA-32 calling standard.

12.1 Address Representation

Probably the most distinguishing feature of the ALPHA-32 flavor of the calling standard is that in **all** cases bits <63:31> of all values that represent addresses are identical. In many cases this fact is used to save storage since only 32 bits are needed to represent an address. Structures that are compatible with the ALPHA-64 flavor of the calling standard allocate a full 64 bits for addresses even though bits <63:31> are always the same.

12.2 Procedure Representation

This flavor of the calling standard represents procedures in the same way as the ALPHA-64 flavor of the standard (i.e. either the address of a statically defined procedure descriptor or, in the case of bound procedures, the address of a dynamically defined procedure descriptor). See Section 3.2, Procedure Representation, and Section 4.5.4, Bound Procedure Values, for more information.

12.3 Register Usage Conventions

This section describes the usage of the ALPHA hardware general purpose (integer) and floating point registers.

12.3.1 Integer Registers

In a standard conforming procedure the general purpose, integer registers are used for the same purposes as in the ALPHA-64 flavor of the standard with the following single exception:

R29	FP, the Frame Pointer. The contents of this register define, among other things, which procedure is considered to be current. Details of usage and alignment are defined in Section 13.3, Procedure Call Chain.

See Section 3.3.1, Integer Registers, for details of other integer register usage.

12.3.2 Floating Point Registers

In a ALPHA-32 standard conforming procedure the floating point registers are used for the same purposes as in the ALPHA-64 flavor of the standard. See Section 3.3.2, Floating Point Registers, for details.

CHAPTER 13

FLOW CONTROL

The following sections contain descriptions of various aspects of the calling standard that deal with the flow of control of a program (as opposed to data manipulation which comes later in Chapter 14).

13.1 Procedure Types

This flavor of the standard has the same three basic types of procedures as the ALPHA-64 flavor. They 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
- No Frame procedure - A procedure that does not establish a context and therefore executes in the context of its caller

See the discussion in Section 4.1, Procedure Types, for more basic information about procedures.

13.1.1 Stack Frame 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 non-intuitive ways such as:

- A called routine may use the stack as a means to return certain types of function values (see Section 14.2.2 for details).
- A called routine that allocates stack may take an exception in its routine prolog before it becomes *current*. In this case the fact that the stack expansion happened in the context of the caller becomes important (see Section 13.4.5, Entry and Exit Code Sequences, for more details).

  It is for this reason that the fixed stack usage version of this procedure type cannot make standard calls.

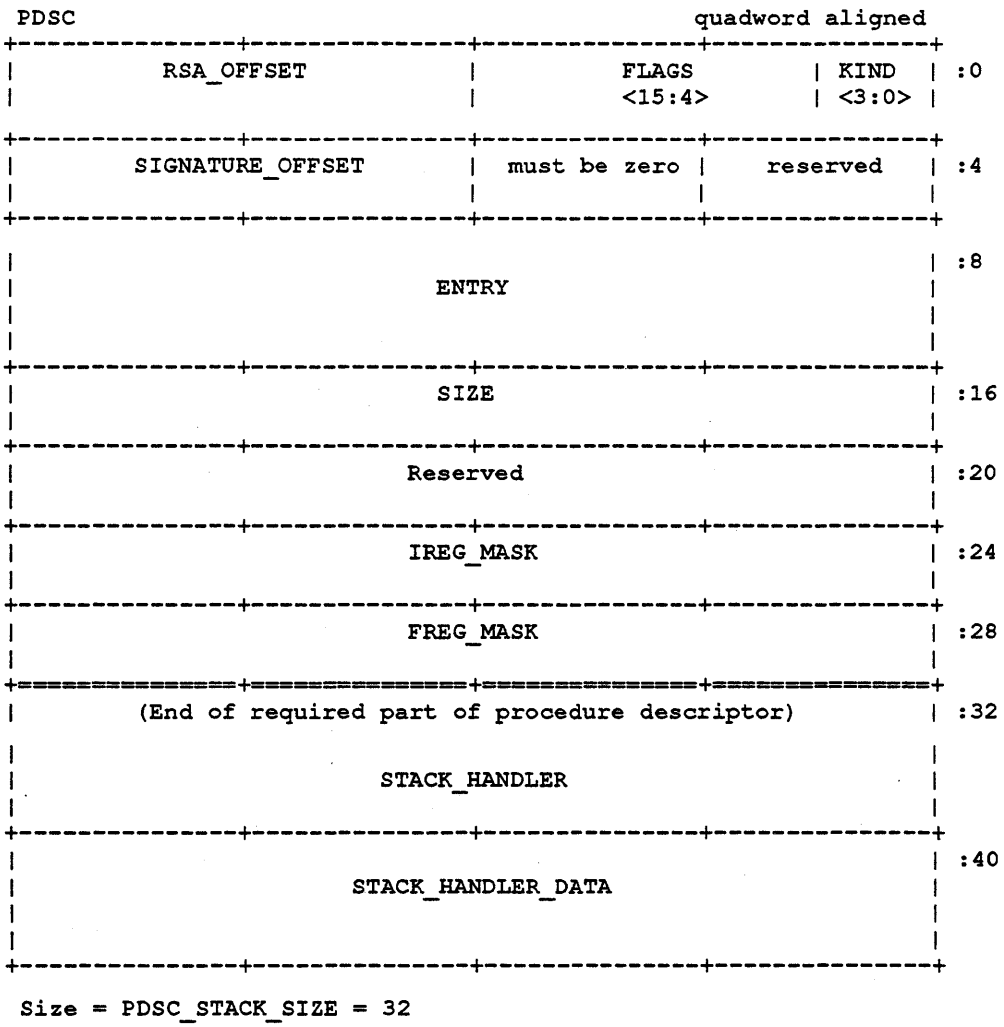
The *variable stack usage* version of this type of procedure is referred to as *full function* and may make standard calls to other procedures.

Procedure Descriptor For Procedures With a Stack Frame

```
\\
Procedure Descriptors for the ALPHA-32 flavor of the calling standard are
designed to be very similar to those in the ALPHA-64 flavor. Even though the
differences are small the entire procedure descriptor layouts are included
below for purposes of clarity. Differences can be easily identified by fields
with FP_ on the front of the field name. All other fields share a common offset
(symbol name/definition) with the ALPHA-64 flavor of the standard.
\\
```

Procedure descriptors for procedures with stack frames are defined as follows:

Figure 13–1: Stack Frame Procedure Descriptor Format



PDSC.KIND = 9 (denoted by PDSC_KIND_FP_STACK) specifies a procedure descriptor for a procedure with a stack frame. The frame base register for such a procedure (SP or FP as indicated by PDSC.FLAGS.BASE_REG_IS_FP) always points to the base of the stack frame (see format below).

PDSC.FLAGS is a vector of flag bits defined as follows.

- PDSC.FLAGS<0> is denoted by HANDLER_VALID.

  If HANDLER_VALID is 1, then PDSC.STACK_HANDLER is present and specifies an exception handler.

- PDSC.FLAGS<1> is denoted by HANDLER_REINVOKABLE.

  HANDLER_REINVOKABLE is 1 if the exception handler is a reinvokable handler, that is, one that can be reinvoked should another exception occur while the handler is already active. If HANDLER_REINVOKABLE is 0 then the exception handler is not reinvokable.

  HANDLER_REINVOKABLE must be 0 unless PDSC.FLAGS.HANDLER_VALID is 1.

  RATIONALE:

  This flag was added to correct a specific behavior that became apparent while implementing run time support using the VAX/VMS exception handling. If an exception arises while another exception handler is already active, VAX/VMS skips all handlers between the active frame and the establisher of the exception handler in which the new exception occurred. BASIC and PL/I have both had to code around this problem. Use of this flag would allow the exception dispatch code to not skip any exception handler that had this bit set.

- PDSC.FLAGS<2> is denoted by HANDLER_DATA_VALID.

  If HANDLER_DATA_VALID is 1, then PDSC.STACK_HANDLER must specify a exception handler, PDSC.FLAGS.HANDLER_VALID must be 1, and PDSC.STACK_HANDLER_DATA must exist. If all of these criteria are met the address of PDSC.STACK_HANDLER_DATA will be passed to the exception handler as defined in Section 15.1, Exception Handling.

- PDSC.FLAGS<3> is denoted by BASE_REG_IS_FP.

  If BASE_REG_IS_FP is 0, then the SP is used as the base register to which PDSC.SIZE is added during an unwind. A procedure with this flag set to 0 has a fixed amount of stack storage specified by PDSC.SIZE, all of which is allocated in the procedure entry sequence, and SP is modified by this procedure only in the entry and exit code sequences. In this case FP normally contains the address of the procedure descriptor for the procedure.

  A procedure that has this flag set to 0 can NOT make standard calls.

  If BASE_REG_IS_FP is 1, then the FP is used as the base. A procedure with this flag set to 1 has a fixed amount of stack storage specified by PDSC.SIZE, and may have a variable amount of stack storage allocated by modifying SP in the body of the procedure.

- PDSC.FLAGS<4> is denoted by REI_RETURN.

  REI_RETURN is 1 if the procedure expects the stack at entry to be set up so that an REI instruction will correctly return from the procedure. If this bit is set the contents of the RSA.SAVED_RETURN field in the register save area are *unpredictable* and the return address is found on the stack. (See Figure 4–4, Register Save Area Layout, for details.)

- PDSC.FLAGS<5> is denoted by STACK_RETURN_VALUE.

STACK_RETURN_VALUE is 1 if the procedure does not reset the stack pointer to its value at procedure entry. This is the case when the function value is returned on the stack.

- PDSC.FLAGS<6> *must* be 0.
- PDSC.FLAGS<7> is denoted by NO_JACKET.

  NO_JACKET must always be set to 1 for compiled code.
- PDSC.FLAGS<8> is denoted by NATIVE.

  NATIVE must always be set to 1 for compiled code.
- PDSC.FLAGS<11:9> *must* be 0.

PDSC.RSA_OFFSET is the signed difference in bytes between the stack frame base (SP or FP as indicated by PDSC.FLAGS.BASE_REG_IS_FP) and the register save area (see layout below). PDSC.RSA_OFFSET must be a multiple of 8 such that (PDSC.RSA_OFFSET) added to the contents of SP or FP, as indicated by PDSC.FLAGS.BASE_REG_IS_FP, yields a quadword-aligned address.

PDSC.SIGNATURE_OFFSET is a 16-bit signed byte offset from the start of the procedure descriptor. This offset, if non-zero, designates the start of the procedure signature block. A zero in this field indicates that no signature information is present. (See Section 4.2, Procedure Signatures, for details of the procedure signature block.)

PDSC.ENTRY is the absolute address of the first instruction of the entry code sequence for the procedure.

PDSC.SIZE is the unsigned size in bytes of the fixed portion of the stack frame for this procedure. The value of SP at entry to this procedure can be calculated by adding PDSC.SIZE to the value SP or FP, as indicated by PDSC.FLAGS.BASE_REG_IS_FP. PDSC.SIZE cannot be 0 for a stack frame type of procedure since the stack frame must include space for the register save area.

PDSC.SIZE *must* be an appropriate multiple to maintain the minimum stack alignment required by the ALPHA hardware architecture. Furthermore, in almost all cases it will be a multiple of 16 to maintain proper stack alignment during a call as defined in Section 4.5.1, Call Conventions.

Various combinations of PDSC.FLAGS.BASE_REG_IS_FP and PDSC.SIZE can be used as follows:

- When PDSC.FLAGS.BASE_REG_IS_FP is 0 and PDSC.SIZE is zero, then the procedure utilizes no stack storage and SP contains the value of SP at entry to the procedure. (Procedure must be a register frame procedure)
- When PDSC.FLAGS.BASE_REG_IS_FP is 0 and PDSC.SIZE is non-zero, then the procedure has a fixed amount of stack storage specified by PDSC.SIZE, all of which is allocated in the procedure entry sequence, and SP is modified by this procedure only in the entry and exit code sequences. (Procedure may not make standard calls)
- When PDSC.FLAGS.BASE_REG_IS_FP is 1 and PDSC.SIZE is non-zero, then the procedure has a fixed amount of stack storage specified by PDSC.SIZE, and may have a variable amount of stack storage allocated by modifying SP in the body of the procedure. (Procedure must be a stack frame procedure)

- The combination when PDSC.FLAGS.BASE_REG_IS_FP is 1 and PDSC.SIZE is zero is illegal since it violates the rules for R29 (FP) usage which say that it must be saved (on the stack) and restored.

PDSC.IREG_MASK 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 R0. (Bits 31, 30, and 28 of this mask should never be set since R31 is the integer Read-As-Zero register, R30 is the hardware SP, and R28 is always assumed to be destroyed during a procedure call or return.)

PDSC.FREG_MASK is a bit vector (0..31) specifying the floating point registers that are saved in 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 since it corresponds to the floating point Read-As-Zero register.)

PDSC.STACK_HANDLER is a signed self-relative pointer to the procedure descriptor for a run time static exception handling procedure. This part of the procedure descriptor is optional. It MUST be supplied if either PDSC.FLAGS.HANDLER_VALID is 1 or PDSC.FLAGS.HANDLER_DATA_VALID is 1 (which requires that PDSC.FLAGS.HANDLER_VALID be 1).

If PDSC.FLAGS.HANDLER_VALID is 0, then the contents or existence of PDSC.STACK_HANDLER is *unpredictable.*

PDSC.STACK_HANDLER_DATA is a quadword, the address of which is passed to the exception handler as context. This is an optional quadword and need only be supplied if PDSC.FLAGS.HANDLER_DATA_VALID is 1.

If PDSC.FLAGS.HANDLER_DATA_VALID is 0, then the contents or existence of PDSC.STACK_HANDLER_DATA is *unpredictable.*

RATIONALE:

Because the address of the handler data cell is passed instead of the data itself, the data may be a self relative pointer.

Stack Frame Format

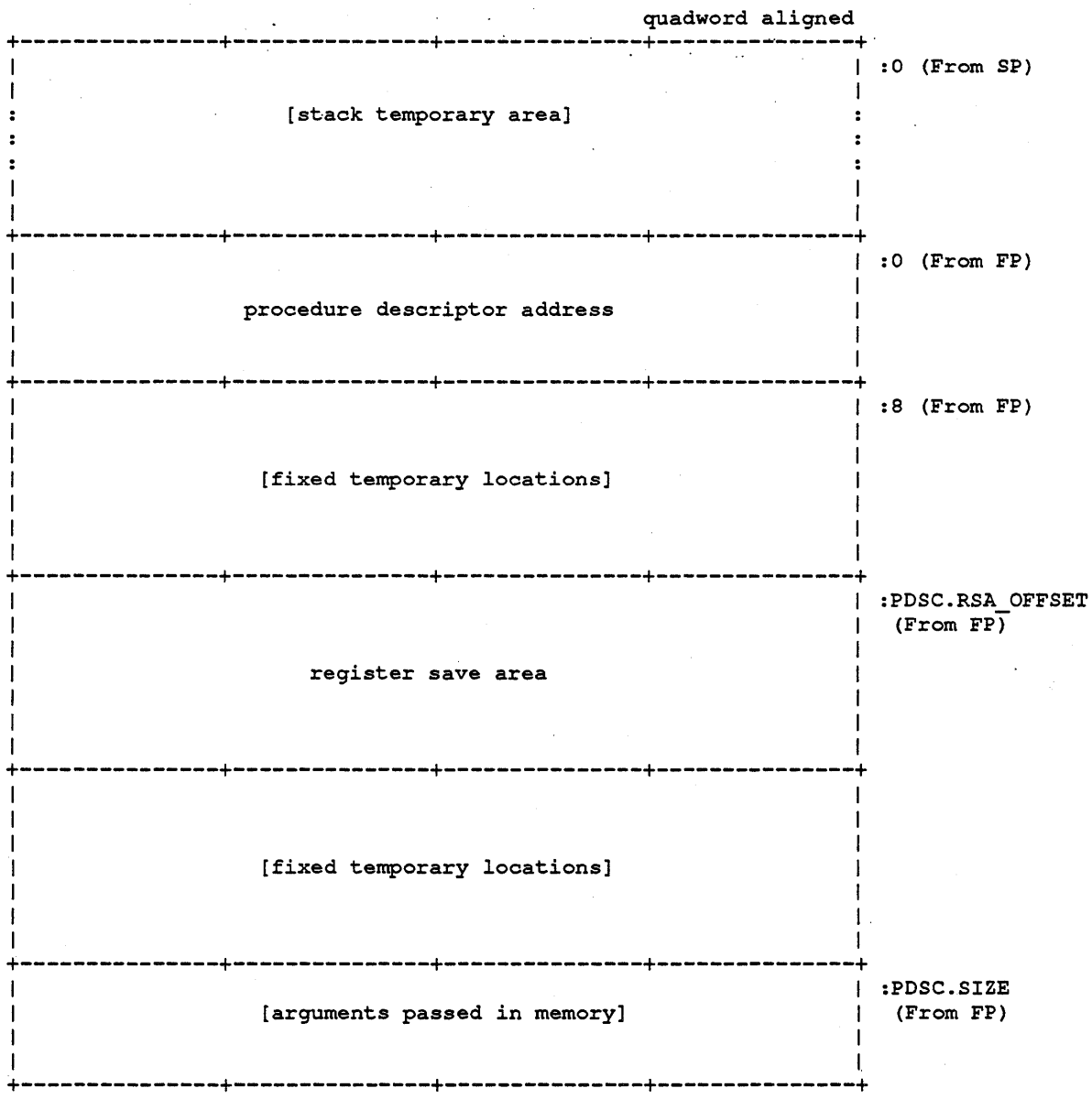
Even though the exact contents of a stack frame are determined by the compiler there are certain properties common to all stack frames. The two basic flavors of stack frames are described below.

The format of the stack frame for a procedure with a fixed amount of stack which uses the SP as the stack base register (i.e. PDSC.FLAGS.BASE_REG_IS_FP is 0) is the same as in the ALPHA-64 flavor of the standard. (See Figure 4–2, Fixed Size Stack Frame Format). In this case, FP would normally contain the address of the procedure descriptor for the procedure.

The following figure illustrates the format of the stack frame for procedures with a varying amount of stack which use the FP as the stack base register (i.e. PDSC.FLAGS.BASE_REG_IS_FP is 1).

Some parts of the stack frame are optional and occur only as required by the particular procedure. Brackets surrounding a field's name indicate the field is optional.

Figure 13–2: Variable Size Stack Frame Format



In either case the portion of the stack frame designated by PDSC.SIZE must be allocated and initialized by the entry code sequence of a called procedure with a stack frame.

Usage of each of the stack areas is the same as in the ALPHA-64 flavor of the standard (see Figure 4–3) with the following exceptions:

The **Procedure descriptor address** at 0(FP) must contain the address of the procedure descriptor of the current procedure if PDSC.FLAGS.BASE_REG_IS_FP is 1.

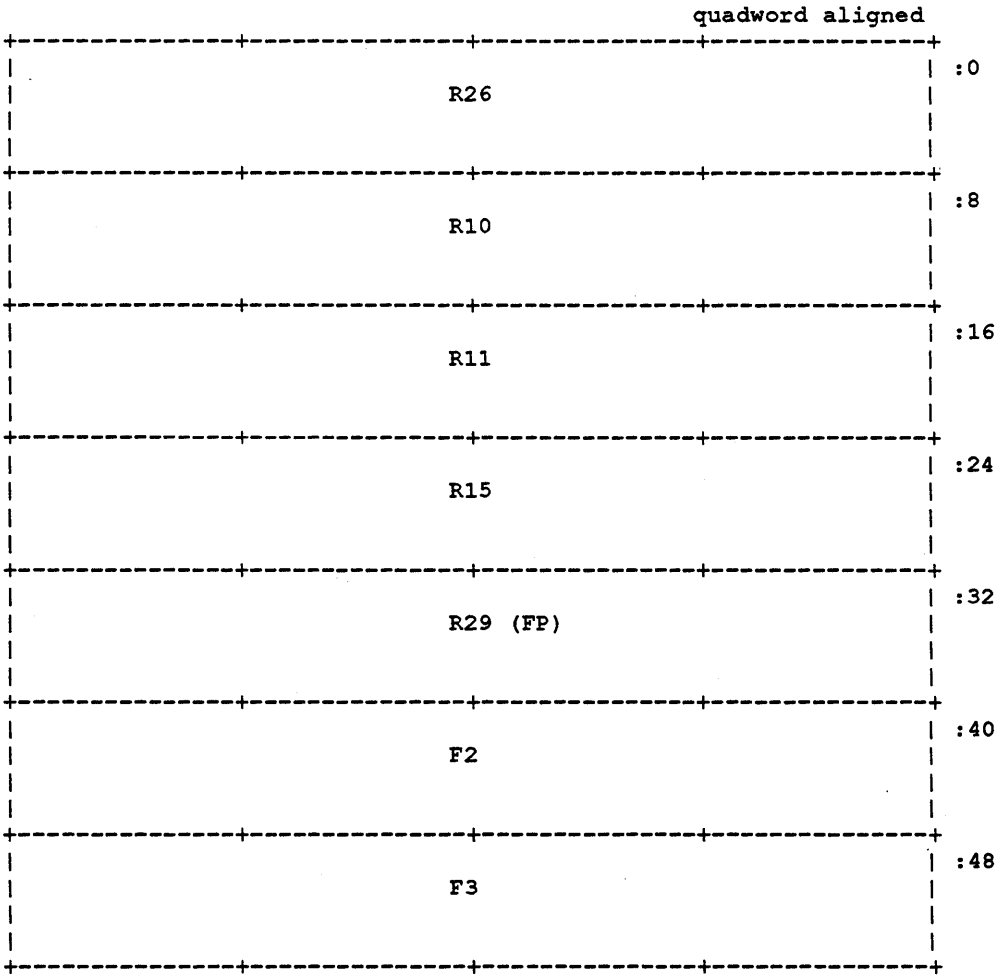
Fixed temporary locations are used in the same way as in the ALPHA-64 flavor of the calling standard including the *argument home area*. In this flavor of the calling standard generally, 6 * 8 bytes of stack storage will be allocated by the called procedure for this purpose.

Register Save Area

The *Register Save Area* is populated and formatted exactly as in the ALPHA-64 flavor of this standard (see Figure 4–4, Register Save Area Layout) with the additional note that the preserved integer register set *must* always include R29 (FP), since it will always be used.

For example, if registers R10, R11, R15, FP, F2, and F3 were to be saved for a procedure called with a standard call, they would be packed in the register save area as follows:

Figure 13–3: Register Save Example



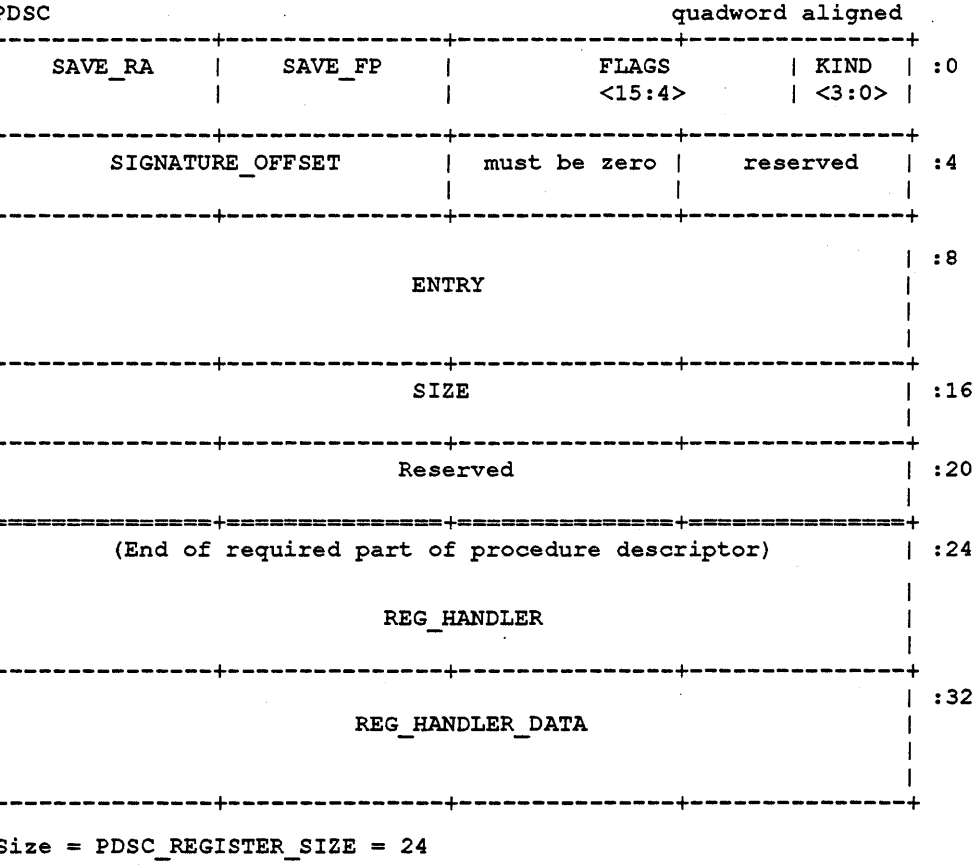
13.1.2 Register Frame Procedure

The considerations and conventions for register frame procedures are the same as in the ALPHA-64 flavor of the calling standard with the exception of the layout of the procedure descriptor defined below.

Procedure Descriptor For Procedures With a Register Frame

Procedure descriptors for procedures with register frames are defined as follows:

Figure 13–4: Register Frame Procedure Descriptor Format



PDSC.KIND = 10 (denoted by PDSC_KIND_FP_REGISTER) specifies a procedure descriptor for a procedure with a register frame.

PDSC.FLAGS is a vector of flag bits defined as follows.

- PDSC.FLAGS<0> is denoted by HANDLER_VALID.

  If HANDLER_VALID is 1, then PDSC.REG_HANDLER must be present and specify an exception handler.

- PDSC.FLAGS<1> is denoted by HANDLER_REINVOKABLE.

  HANDLER_REINVOKABLE is 1 if the exception handler is a reinvokable handler, that is, one that can be reinvoked should another exception occur while the handler is already active. If HANDLER_REINVOKABLE is 0 then the exception handler is not reinvokable.

  HANDLER_REINVOKABLE must be 0 unless PDSC.FLAGS.HANDLER_VALID is 1.

- PDSC.FLAGS<2> is denoted by HANDLER_DATA_VALID.

If HANDLER_DATA_VALID is 1, then PDSC.REG_HANDLER must specify a exception handler, PDSC.FLAGS.HANDLER_VALID must be 1, and PDSC.REG_HANDLER_DATA must exist. If all these criteria are met the address of PDSC.REG_HANDLER_DATA will be passed to the exception handler as described in Section 15.1, Exception Handling.

- PDSC.FLAGS<3> is denoted by BASE_REG_IS_FP.

  If BASE_REG_IS_FP is 0, then the SP is used as the base register to which PDSC.SIZE is added during an unwind. A procedure with this flag set to 0 has a fixed amount of stack storage specified by PDSC.SIZE, all of which is allocated in the procedure entry sequence, and SP is modified by this procedure only in the entry and exit code sequences. In this case FP normally contains the address of the procedure descriptor for the procedure.

  A procedure that has this flag set to 0 can NOT make standard calls.

  If BASE_REG_IS_FP is 1, then the FP is used as the base and must point to a stack location that contains the address of the procedure's descriptor. A procedure with this flag set to 1 has non-zero fixed amount of stack storage specified by PDSC.SIZE, and may have a variable amount of stack storage allocated by modifying SP in the body of the procedure.

- PDSC.FLAGS<4> is denoted by REI_RETURN.

  REI_RETURN is 1 if the procedure expects the stack at entry to be set up so that an REI instruction will correctly return from the procedure. If this bit is set the contents of the PDSC.SAVE_RA field are *unpredictable* and the return address is found on the stack.

- PDSC.FLAGS<5> is denoted by STACK_RETURN_VALUE.

  STACK_RETURN_VALUE is 1 if the procedure does not reset the stack pointer to its value at procedure entry. This is the case when the function value is returned on the stack.

- PDSC.FLAGS<6> *must* be 0.

- PDSC.FLAGS<7> is denoted by NO_JACKET.

  NO_JACKET must always be set to 1 for compiled code.

- PDSC.FLAGS<8> is denoted by NATIVE.

  NATIVE must always be set to 1 for compiled code.

- PDSC.FLAGS<11:9> *must* be 0.

PDSC.SAVE_FP is the number of the register in which the value of FP at entry to this procedure is maintained.

In a standard procedure, PDSC.SAVE_FP must specify a scratch register so as not to violate the rules for procedure entry code as specified in Section 13.4.5.

PDSC.SAVE_RA is the number of the register in which the return address is maintained. If this procedure uses the standard call conventions and does not modify R26, then PDSC.SAVE_RA can specify R26.

In a standard procedure, PDSC.SAVE_RA must specify a scratch register so as not to violate the rules for procedure entry code as specified in Section 13.4.5.

PDSC.SIGNATURE_OFFSET is a 16-bit signed byte offset from the start of the procedure descriptor. This offset, if non-zero, designates the start of the procedure signature block. A zero in this field indicates that no signature information is present. (See Section 4.2, Procedure Signatures, for details of the procedure signature block.)

PDSC.ENTRY is the absolute address of the first instruction of the entry code sequence for the procedure.

PDSC.SIZE is the unsigned size in bytes of the fixed portion of the stack storage (if any) for this procedure. The value of SP at entry to this procedure can be calculated by adding PDSC.SIZE to the value of the register indicated by PDSC.FLAGS.BASE_REG_IS_FP. If a procedure uses no stack storage then PDSC.SIZE is 0.

PDSC.SIZE *must* be an appropriate multiple to maintain the minimum stack alignment required by the ALPHA hardware architecture. Furthermore, in almost all cases it will be a multiple of 16 to maintain proper stack alignment during a call as defined in Section 4.5.1, Call Conventions.

PDSC.REG_HANDLER is a signed self-relative pointer to the procedure descriptor for a run time static exception handling procedure. This part of the procedure descriptor is optional. It MUST be supplied if either PDSC.FLAGS.HANDLER_VALID is 1 or PDSC.FLAGS.HANDLER_DATA_VALID is 1 (which requires that PDSC.FLAGS.HANDLER_VALID be 1).

If PDSC.FLAGS.HANDLER_VALID is 0, then the contents or existence of PDSC.REG_HANDLER is *unpredictable.*

PDSC.REG_HANDLER_DATA is a quadword the address of which is passed to the exception handler as context. This is an optional quadword and need only be supplied if PDSC.FLAGS.HANDLER_DATA_VALID is 1.

If PDSC.FLAGS.HANDLER_DATA_VALID is 0, then the contents or existence of PDSC.REG_HANDLER_DATA is *unpredictable.*

RATIONALE:

Because the address of the handler data cell is passed instead of the data itself, the data may be a self relative pointer.

13.1.3 No Frame Procedure

The considerations and format of the procedure descriptor for a *no frame* procedure are the same as in the ALPHA-64 flavor of the calling standard. See Section 4.1.3, No Frame Procedure, for details.

13.2 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 as defined in Appendix B, Procedure Descriptor Access Routines.

13.3 Procedure Call Chain

13.3.1 Current Procedure

In the ALPHA-32 flavor of the calling standard the frame pointer register (R29, also referred to as FP) is used to indicate the *current procedure.*

To accomplish this the *current procedure* must **always** maintain in FP one of the following:

- A pointer to the procedure descriptor for that procedure

  or

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

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

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

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

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

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

        ;Error - Invalid FP

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

        ;Procedure KIND is now in R0
```

If kind is equal to PDSC_KIND_FP_STACK then the current procedure has a stack frame.

If kind is equal to PDSC_KIND_FP_REGISTER, the current procedure is a register frame procedure.

Note that either type of procedure may use either type of mechanism to point to the procedure descriptor. Compilers may make the appropriate choice of which mechanism to use based on the needs of the procedure involved.

13.3.2 The Call Chain

The definition of the call chain and its usage is the same for this flavor of the calling standard as in the ALPHA-64 flavor of the standard (see Section 4.4.2, The Call Chain, for details).

13.3.3 Procedure Call Tracing

The considerations and conventions for tracing the call chain are the same for this flavor of the calling standard as in the ALPHA-64 flavor of the standard. See Section 4.4.3, Procedure Call Tracing, for details.

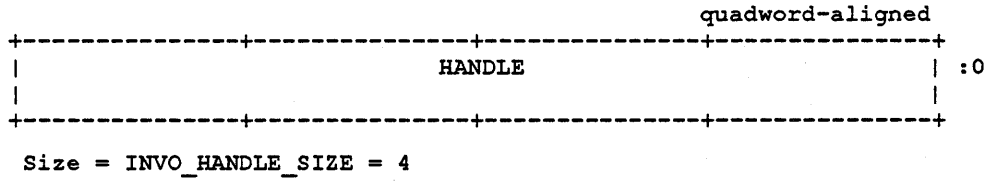
13.3.4 Obtaining the Context of a Procedure

The considerations and conventions for obtaining a procedure context are the same for this flavor of the calling standard as in the ALPHA-64 flavor of the standard with the following single exception:

- In the ALPHA-32 flavor of the calling standard, a procedure invocation handle only consists of 32 bits.

This handle is defined as follows:

Figure 13–5: Procedure Invocation Handle Format



HANDLE is a 32-bit quantity which may be used to refer to a specific procedure invocation at run time. (See Section D.2.1, Invocation Handles, for how this handle might be constructed.)

See Section 4.4.4, Obtaining the Context of a Procedure, for other details of invocation referencing.

13.4 Transfer of control

The considerations and conventions for transferring control are the same for this flavor of the calling standard as in the ALPHA-64 flavor of the standard. See Section 4.5, Transfer of control, for details.

13.4.1 Call Conventions

The call conventions for the ALPHA-32 flavor of the calling standard are the same as for the ALPHA-64 flavor of the standard (See Section 4.5.1, Call Conventions, for details) with the following exception:

- **Function Results**

  In the ALPHA-32 flavor of the calling standard descriptors that are used for function returns are formated in a VAX/VMS compatible way. See Section 14.2.2, Returning Data for details.

13.4.2 Linkage

The linkage considerations and conventions are the same for this flavor of the calling standard as in the ALPHA-64 flavor of the standard. See Section 4.5.2, Linkage, for details.

13.4.3 Calling Computed Addresses

The considerations and conventions for calling computed addresses are the same for this flavor of the calling standard as in the ALPHA-64 flavor of the standard. See Section 4.5.3, Calling Computed Addresses, for details.

13.4.4 Bound Procedure Values

The considerations and conventions for bound procedure values, including the format of the bound procedure descriptor, are the same for this flavor of the calling standard as in the ALPHA-64 flavor of the standard. See Section 4.5.4, Bound Procedure Values, for details.

13.4.5 Entry and Exit Code Sequences

To ensure that the stack can be interpreted at any point during thread execution, all procedures must adhere to certain conventions for entry and exit. These conventions are described below.

Entry Code Sequence

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

- All registers specified by this standard as saved across a standard call must contain their original (at entry) contents.

- No standard calls may be made

NOTE

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

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

This is done by performing the following actions:

Do step 1, then steps 2 - 7 in any order, then step 8 or 9.

1. If PDSC.SIZE is not 0, set register SP = SP - PDSC.SIZE.
2. If PDSC.FLAGS.BASE_REG_IS_FP is 1, store the address of the procedure descriptor at 0(SP)
3. If PDSC.KIND = PDSC_KIND_FP_REGISTER, copy the return address to the register specified by PDSC.SAVE_RA if it is not already there.
4. If PDSC.KIND = PDSC_KIND_FP_REGISTER, copy the FP register to the register specified by PDSC.SAVE_FP.
5. If PDSC.KIND = PDSC_KIND_FP_STACK, copy the return address to the quadword at PDSC.RSA_OFFSET.SAVED_RETURN.
6. If PDSC.KIND = PDSC_KIND_FP_STACK store the registers specified by PDSC.IREG_MASK and PDSC.FREG_MASK in the register save area as denoted by PDSC.RSA_OFFSET. (This step includes saving the value in FP.)
7. Execute DRAINT if required (see Section 15.1.6, Other Considerations, for details)
8. If PDSC.FLAGS.BASE_REG_IS_FP is 1, load register FP with the address of the procedure descriptor or the address of a quadword that contains the address of the procedure descriptor.
9. If PDSC.FLAGS.BASE_REG_IS_FP is 1, copy register SP to register FP.

Entry Code Example for a Stack Frame

This example assumes that this is a stack frame procedure, that registers R2..R4 and F2..F3 are saved and restored, that PDSC.RSA_OFFSET = 16, that the procedure has a static exception handler that does not reraise arithmetic traps, and that the procedure uses a variable amount of stack.

```
LDA     SP,-SIZE(SP)    ;Allocate space for new stack frame.
STQ     R27,(SP)        ;Set up address of procedure descriptor
STQ     R26,16(SP)      ;Save return address
STQ     R2,24(SP)       ;Save first integer register.
STQ     R3,32(SP)       ;Save next integer register
STQ     R4,40(SP)       ;Save next integer register
STQ     FP,48(SP)       ;Save caller's frame pointer
STT     F2,56(SP)       ;Save first floating point register
STT     F3,64(SP)       ;Save last floating point register
DRAINT                  ;Force any pending hardware exceptions to be
                        ; raised. (see Section 15.1.6)
MOV     SP,FP           ;Called procedure is now the current procedure.
```

Note that if this code sequence is interrupted by an asynchronous software interrupt, SP will have a different value than it did at entry, but the calling procedure will still be current.

At that point, it would not be possible to determine the original value of SP by the register frame conventions. If actions by an exception handler result in a nonlocal GOTO to a location in the immediate caller, then it will not be possible to restore SP to the correct value in that caller.

Therefore, any procedure that contains a label that can be the target of a non-local GOTO by immediately called procedures must be prepared to reset or otherwise manage the SP at that label.

Entry Code Example for a Register Frame

This example assumes that the called procedure has no static exception handler, utilizes no stack storage, PDSC.SAVE_RA specifies R26, PDSC.SAVE_FP specifies R22, and PDSC.FLAGS.BASE_REG_IS_FP is 0.

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

Exit Code Sequence

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

This is done by performing the following actions:

Do steps 1 - 4 in any order, then the remaining steps in the order specified.

1. If PDSC.FLAGS.BASE_REG_IS_FP is 1, then copy FP to SP
2. If PDSC.KIND = PDSC_KIND_FP_STACK, and this procedure saves/restores any registers *other than* FP and SP, reload those registers from the register save area as specified by PDSC.RSA_OFFSET.
3. If PDSC.KIND = PDSC_KIND_FP_STACK, load a scratch register with the return address from the register save area as specified by PDSC.RSA_OFFSET. (If PDSC.KIND = PDSC_KIND_FP_REGISTER, the return address is already in scratch register PDSC.SAVE_RA.)
4. Execute DRAINT if required (see Section 15.1.6, Other Considerations, for details)
5. If PDSC.KIND = PDSC_KIND_FP_REGISTER, copy the register specified by PDSC.SAVE_FP to register FP.
6. If PDSC.KIND = PDSC_KIND_FP_STACK, reload FP from the saved FP in the register save area.
7. If a function value is not being returned at 0(SP), then restore SP to the value it had at procedure entry by adding the value that was stored in PDSC.SIZE to SP. (In some cases the returning procedure will leave SP pointing to a lower stack address than it had on entry to the procedure, as specified in Section 14.2.2, Returning Data).
8. Jump to the return address (which is in a scratch register).

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 may choose to defer their removal due to optimizations or other considerations.

Exit Code Example for a Stack Frame

The following is the return code sequence for the stack frame example above:

```
MOV     FP,SP           ;Chop the stack back
LDQ     R28,16(FP)      ;Get return address
LDQ     R2,24(FP)       ;Restore first integer register.
LDQ     R3,32(FP)       ;Restore next integer register
LDQ     R4,40(FP)       ;Restore next integer register
LDT     F2,56(FP)       ;Restore first floating point register
LDT     F3,64(FP)       ;Restore last floating point register
DRAINT                  ;Force any pending hardware exceptions to be
                        ; raised.  (see Section 15.1.6)
LDQ     FP,48(FP)       ;Restore caller's frame pointer
LDA     SP,SIZE(SP)     ;Restore SP (SIZE is compiled into PDSC.SIZE)
RET     R31,(R28)       ;Return to caller's code.
```

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

Exit Code Example for a Register Frame

The following is the return code sequence for the register frame example above:

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

CHAPTER 14

DATA MANIPULATION

This section deals with the storage and passing of data.

14.1 Data Types

14.1.1 Argument and Function Value Data Types

```
\\
The detailed description of the ALPHA data types may be included here later.
For now, section "C.7 Argument Data Types" of the VAX Procedure
Calling and Condition Handling Standard, Revision 10.3 may be used as a
reference.

The IEEE data types are not currently defined in the document mentioned above.
Their format can be found in the ALPHA SRM.
\\
```

Fully Supported Data Types

The following VAX data types are fully supported for use as standard argument data types, and may be passed between external procedures and between languages.

These data types may be freely utilized in the either the ALPHA-32 or ALPHA-64 environments.

Table 14–1: Fully Supported Data Types

Atomic Data Type	Type Designator	Natural Alignment
byte logical	BU	byte
word logical	WU	word
longword logical	LU	longword
quadword logical	QU	quadword
byte integer	B	byte
word integer	W	word
longword integer	L	longword
quadword integer	Q	quadword
F floating	F	longword
D floating	D	quadword
G floating	G	quadword
F floating complex	FC	longword
D floating complex	DC	quadword
G floating complex	GC	quadword
IEEE floating basic single S	FS	longword
IEEE floating basic double T	FT	quadword
IEEE floating basic single S complex	FSC	longword
IEEE floating basic double T complex	FTC	quadword

String Data Type	Type Designator	Natural Alignment
character coded text	T	byte
varying character coded text	VT	word
numeric string, unsigned	NU	byte
numeric string, left separate sign	NL	byte
numeric string, left overpunched sign	NLO	byte
numeric string, right separate sign	NR	byte
numeric string, right overpunched sign	NRO	byte
numeric string, zoned sign	NZ	byte

Miscellaneous Data Type	Type Designator	Natural Alignment
descriptor	DSC	longword
bound label value	BLV	longword
absolute date and time	ADT	quadword

The following data types are supported for compatibility reasons only:

String Data Type	Type Designator	Natural Alignment
two-byte character coded text	T2	word
varying two-byte character coded text	VT2	word

RATIONALE:

The VAX/VMS calling standard architect says T2 and VT2 were never documented or used, even in Asia. Thus they are not supported in the ALPHA-64 environment. They will be removed here if they are dropped from the VAX/VMS calling standard.

Data Types without Robust Hardware Support

The following VAX data types are supported for use as standard argument data types, and may be passed between external procedures and between languages.

These data types may be utilized but it should be noted that the VAX hardware architecture provides special support for these data types which the ALPHA hardware does not provide. Therefore, language or application utilization of these VAX data types may result in (sometimes severe) performance penalties caused by software handling of these types.

Table 14–2: Data Types without Robust Hardware Support

Atomic Data Type	Type Designator	Natural Alignment
octaword logical	OU	quadword
octaword integer	O	quadword

String Data Type	Type Designator	Natural Alignment
bit	V	byte
bit unaligned	VU	N/A
packed decimal	P	byte

Unsupported Data Types

The following VAX data types are unsupported for use as standard argument data types, and should not be passed between standard external procedures or between languages.

Table 14–3: Unsupported Data Types

Atomic Data Type	Type Designator	Reason
COBOL Intermediate Temporary	CIT	VAX COBOL specific
H floating[1]	H	Not supported by Hardware
H floating complex[1]	HC	Not supported by Hardware

[1]H_floating data is not supported for general usage in either flavor of the ALPHA calling standard. However, conversion routines will be supplied to allow users to convert existing H_floating data to other storage formats.

Miscellaneous Data Type	Type Designator	Reason
sequence of instructions	ZI	This data type represents VAX instructions
procedure entry mask	ZEM	ALPHA procedures have no entry mask.
VAX bound procedure value	BPV	Replaced by the mechanism defined in Section 4.5.4, Bound Procedure Values

14.1.2 Argument Descriptors

ALPHA-32 descriptors are defined by reference to Section "Argument Descriptor Formats" of the *VAX Procedure Calling and Condition Handling Standard, Revision 10.3*. That document should be consulted for complete descriptions of the descriptors discussed here.

Compatible Descriptor Classes

The following VAX descriptor classes are supported for use as standard ALPHA-32 environment argument descriptors, and may be passed between external procedures and between languages with exactly the same rules and semantics as the corresponding VAX/VMS descriptors.

Table 14–4: Compatible Descriptor Classes

Descriptor Class	Class Designator
Scalar string	S
Dynamic string	D
Array	A
Decimal scalar string	SD
Noncontiguous array	NCA
Varying string	VS
Varying string array	VSA
Unaligned bit string	UBS
Unaligned bit array	UBA
String with bounds	SB
Unaligned bit string with bounds	UBSB

Adjusted Descriptor Classes

The following VAX/VMS descriptor classes are supported for use as standard ALPHA-32 environment argument descriptors, and may be passed between external procedures and between languages. Their contents and/or semantics, however, are different from those of VAX/VMS as described below.

Table 14–5: Adjusted Descriptor Classes

Descriptor Class	Class Designator	Adjustment
Procedure descriptor	P	The DSC$A_POINTER field must specify the address of a procedure descriptor for the procedure, *not* the address of the entry mask to the procedure.

Unsupported Descriptor Classes

The following VAX descriptor classes are unsupported for use as standard ALPHA-32 environment argument data types, and should not be passed between standard external procedures or between languages.

Table 14–6: Unsupported Descriptor Classes

Descriptor Class	Class Designator	Reason
Variable buffer descriptor	V	Digital Internal Use Only
Procedure Invocation descriptor	PI	Obsolete
Label descriptor	J	Reserved to VAX Debugger
Label incarnation descriptor	JI	Obsolete
BASIC file array descriptor	BFA	Digital Internal Use Only, may not be passed between separately compiled modules on VAX.

14.1.3 Status Codes and Exception Values

Status codes and exception values for the ALPHA-32 environment are defined in Section "Condition Value" of the *VAX Procedure Calling and Condition Handling Standard, Revision 10.3*.

14.2 Data Passing

Most of the conventions and considerations for data passing in the ALPHA-32 flavor of the calling standard are the same as for the ALPHA-64 flavor (See Section 5.2, Data Passing, for details).

There are, however, a small number of differences that are described below.

14.2.1 Argument Passing Mechanisms

The mechanisms used to pass data are the same as the ALPHA-64 flavor, namely:

- Immediate value
- Reference
- Descriptor

The considerations for the mechanism usage are the same with the following additional notes:

- Reference argument items must be sign extended to 64 bits.
- Descriptors used are VAX/VMS (32-bit address) style descriptors
- Descriptor argument items (the address of the descriptor) must be sign extended to 64 bits.

14.2.2 Returning Data

The function value return mechanisms are the same as for the ALPHA-64 flavor of the calling standard, namely:

- immediate value
- reference
- descriptor

The considerations for returning data by immediate Value and by reference are the same as the ALPHA-64 flavor of the calling standard.

Function Value Return By Descriptor

A function value is returned by *Descriptor* if, and *only* if,

- the actual size of the function value is not known to both the calling procedure and the called procedure and the value cannot be returned by immediate value (because the function value requires more than 64 bits, the data type is a string or an array type, etc.)

  and

- the function value can be returned in a contiguous region of storage.

Non-contiguous function values are language-specific and can not be returned as a standard conforming return value.

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

There are two distinct cases where a descriptor return value is used. They are:

- When the caller determines the maximum size of the returned value
- When the called routine determines the maximum size of the returned value

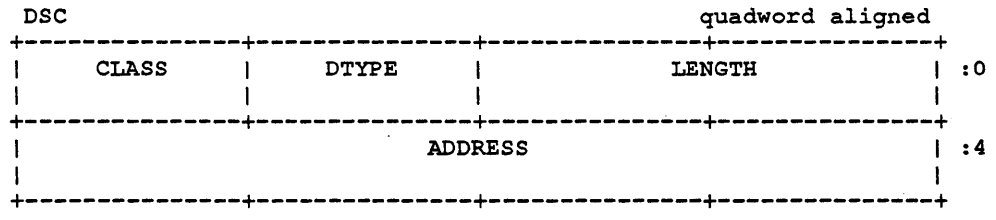
In each case the actual-argument list and the formal-argument list are shifted to the right by one argument item. The new, first argument item is reserved for the function value. This *hidden* first argument is included in the count and register usage information that is passed in the argument information register.

The two cases of function value return by descriptor are described below. The mechanism to be used is chosen by the calling procedure, and is specified by the kind of the descriptor passed as the first argument item.

Caller Defined Return By Descriptor

When the calling procedure determines the maximum size of the returned value it must provide the required contiguous storage and pass the address of a naturally aligned descriptor that describes that storage as the first argument. This descriptor is laid out as follows:

Figure 14–1: ALPHA-32 Caller Defined Return Value Descriptor Format



The calling routine must initialize the descriptor as follows:

- DSC.CLASS must be set to DSC$K_CLASS_S
- DSC.ADDRESS must be set to point to the buffer where the value is to be returned
- DSC.LENGTH must be set to reflect the size of the buffer

To return the value the called function must:

- Write the return value into the storage specified by the hidden descriptor argument, truncating the return value if its length exceeds the value of DSC.LENGTH.
- Provide whatever padding is required by the semantics of the language of the called function.

```
\\
It should be noted that various languages have different language requirements.
FORTRAN and PASCAL, for example, require BLANK padding while Ada does
not pad at all.

This standard makes no attempt to reconcile these differences.
\\
```

The descriptor may be allocated in read-only storage by the caller and thus the called function must never modify the descriptor.

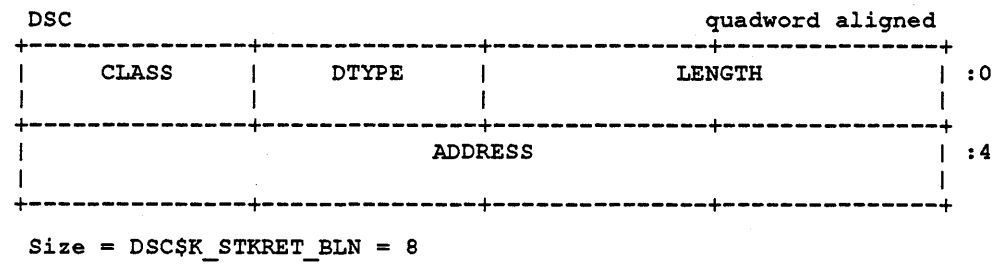
RATIONALE:

This type of function value return is specifically for FORTRAN.

Called Routine Defined Return By Descriptor

When the called routine determines the maximum size of the returned value the stack is used to return the actual data of the value. In this case the descriptor pointed to by the hidden first argument is used as follows:

Figure 14–2: ALPHA-32 Called Routine Defined Return Value Descriptor Format



The caller must pass as the first argument a function return descriptor initialized as follows:

DSC.CLASS = DSC$K_CLASS_STKRET = [TBS]
DSC.DTYPE = As appropriate
DSC.ADDRESS = *unpredictable*
DSC.LENGTH = 0

The descriptor must be allocated in writeable storage, and will be modified by the called function.

The called function must return its value as follows:

1. Allocate stack storage sufficient to contain the return value and place the return value on the stack.
2. Update the descriptor as follows:

   DSC.CLASS = DSC$K_CLASS_STKRET
   DSC.DTYPE = As appropriate
   DSC.ADDRESS = the address of the first byte of the return value
   DSC.LENGTH = an integer $0..2^{16} - 1$ specifying the length in bytes of the return value

   NOTE

   In all cases DSC.ADDRESS must be set before DSC.LENGTH is made non zero. This allows unwind operations to properly restore the stack pointer.

3. The return code sequence must not reset SP such that any part of the function return value is contained by a stack address lower than 0(SP). The return value must be entirely contained by stack storage at or above 0(SP).

When control returns to the calling procedure, the contents of register R0 are *unpredictable*. The caller must manage the return value and SP. The caller may copy the return value from the stack to some other storage (possibly to a higher address on the stack) and reset SP appropriately to reflect the return from the called function.

Run Time Conventions for Function Value Return Descriptors

Any standard-conforming function that supports returning values using the DSC$K_CLASS_STKRET type of descriptor must also be prepared to handle the other descriptor return value mechanism, and must return its value according to the mechanism chosen by the caller.

Functions that may return a stack function return value *must* use the following algorithm to determine which of the two mechanisms to use:

1. If DSC.CLASS = DSC$K_CLASS_S, then use the fixed buffer mechanism.
2. Otherwise, use the stack return mechanism.

It is important that functions do not specifically test for DSC.CLASS = DSC$K_CLASS_STKRET. This is because future extensions to this standard may specify new function return mechanisms that define new CLASS codes, but that are upward compatible with the stack return mechanism specified here.

```
\\
The case in mind here is return via heap storage allocated by the called
function, in which case the caller could "request" return in such heap storage,
but a called function could at its own option override that request and use the
stack return mechanism instead, by changing DSC.CLASS in the passed descriptor.
Callers that "request" return in heap storage would have to be prepared to
handle either heap or stack return from the called function.
\\
```

Any function that tests for DSC.CLASS = DSC$K_CLASS_STKRET may not be upward compatible with future versions of this standard.

For the same reason, functions that return a value by the stack return mechanism must *always* set DSC.CLASS = DSC$K_CLASS_STKRET. Any function that returns a value by stack return mechanism and does not set DSC.CLASS = DSC$K_CLASS_STKRET may not be upward compatible with future versions of this standard.

14.3 Static Data

The conventions and considerations around the use of static data including alignment and record format conventions are the same for this flavor of the calling standard as the 64-bit flavor. See the discussion in Section 5.3, Static Data, for details.

CHAPTER 15

EVENT PROCESSING

This chapter discusses specifications related to events that are normally outside the normal program flow.

15.1 Exception Handling

This section on exception handling discusses the considerations involved in the notification and handling of exceptional events during the course of normal program execution.

Exception notification and handling is closely tied to the hardware mechanisms for exceptional event reporting. The ALPHA architecture is notably different from a conventional VAX in this area.

```
\\
The basic designs and mechanisms of ALPHA Exception handling and unwinding are
modeled after the VAX/VMS design with extensions and corrections to support
POSIX and multithread environments better. That design is defined in Sections
"VAX Conditions", "Operations Involving Condition Handlers", "Properties of
Condition Handlers", and "Multiple Active Signals" of the VAX
Procedure Calling and Condition Handling Standard, Revision 10.3 which may
serve as useful background reading.
\\
```

15.1.1 Exception Handling Requirements

The exception handling requirements for the ALPHA-32 flavor of the calling standard are simply to provide the semantics and functionality of the VAX/VMS exception handling.

15.1.2 Exception Handling Overview

Basic workings

The workings of the exception handling mechanisms for the ALPHA-32 flavor of the calling standard are described in various places in the VAX/VMS documentation set (but referred to as *condition* handling).

15.1.3 Exception Handlers

The types of exception handlers for the ALPHA-32 flavor of the calling standard are the same as for VAX/VMS. They are:

- *Primary* handler
- *Secondary* handler
- *Frame-based* handlers
- *Last-chance* handler
- The system *catchall* handler

See the VAX/VMS documentation for a description of each of these types of handlers and how they are used.

Other Properties of Exception Handlers

Access to Memory
The ALPHA hardware presents certain differences in memory usage when compared with conventional VAX hardware. See the discussion in Section 6.1.4, Exception Handlers, for details.

15.1.4 Establishing and Revoking an Exception Handler

Dynamic activation and deactivation of exception handlers is not defined by this calling standard (and in fact 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 which choose to support this functionality may set up language-specific static exception handlers that provide the dynamic exception handling semantics of that language. These 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 (i.e. calls to *LIB$ESTABLISH* or other dynamic activation routines). This requirement stems from the need to add appropriate DRAINT instructions and other compile time considerations needed to make dynamic exception handling function correctly.

```
\\
There may be additional protocols and conventions for dynamic exception
handling. These may be needed, for example, to enable DEBUG to do a good job
within the language exception handling environment.  These conventions will be
driven by the requirements of the languages and the language support utilities,
and will not be addressed by the calling standard.
\\
```

15.1.5 Raising Exceptions

Raising Unwind Exceptions

See Section 15.2, Unwinding, for details.

Raising Signal Exceptions

Signal exceptions may 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 notifications such as the POSIX defined *alarm()* function. Complete exposition of these circumstances is beyond the scope of this document.

A variety of run time support packages may chose the signal mechanism as a means to provide certain features. One such package is the ANSI C standard support package which provides the *raise()* function to enable a programmer raise a synchronous signal exception (defined as *raise(int sig)* - see *Draft Proposed American National Standard for Information Systems Programming Language C* section 4.7.2.1 for details). Another such package is the POSIX support package which provides the *kill()* procedure. The basic parameters of this function are provided below as an example of one means of raising a signal exception.

A thread may raise a signal exception in its own or another process's context by calling a POSIX library routine defined as follows:

kill(pid, sig)

Arguments:

pid_t pid;	The process identification of the process to receive the signal.
int sig;	The signal value to be delivered.

Function Value:

0	Success
-1	Failure

```
\\
The above is presented simply to convey the basic ideas of raising a
signal. IEEE standard 1003.1-1988 section 3.3.2 should be referenced
for complete details of the workings of this call.
\\
```

Raising General Exceptions

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

LIB$SIGNAL(EXCEPTION_VALUE, ...)

See the VAX/VMS documentation for details.

15.1.6 Other Considerations

Exception Synchronization

See the discussion in Section 6.1.11, Other Considerations.

Continuation from Exceptions

See the discussion in Section 6.1.11, Other Considerations.

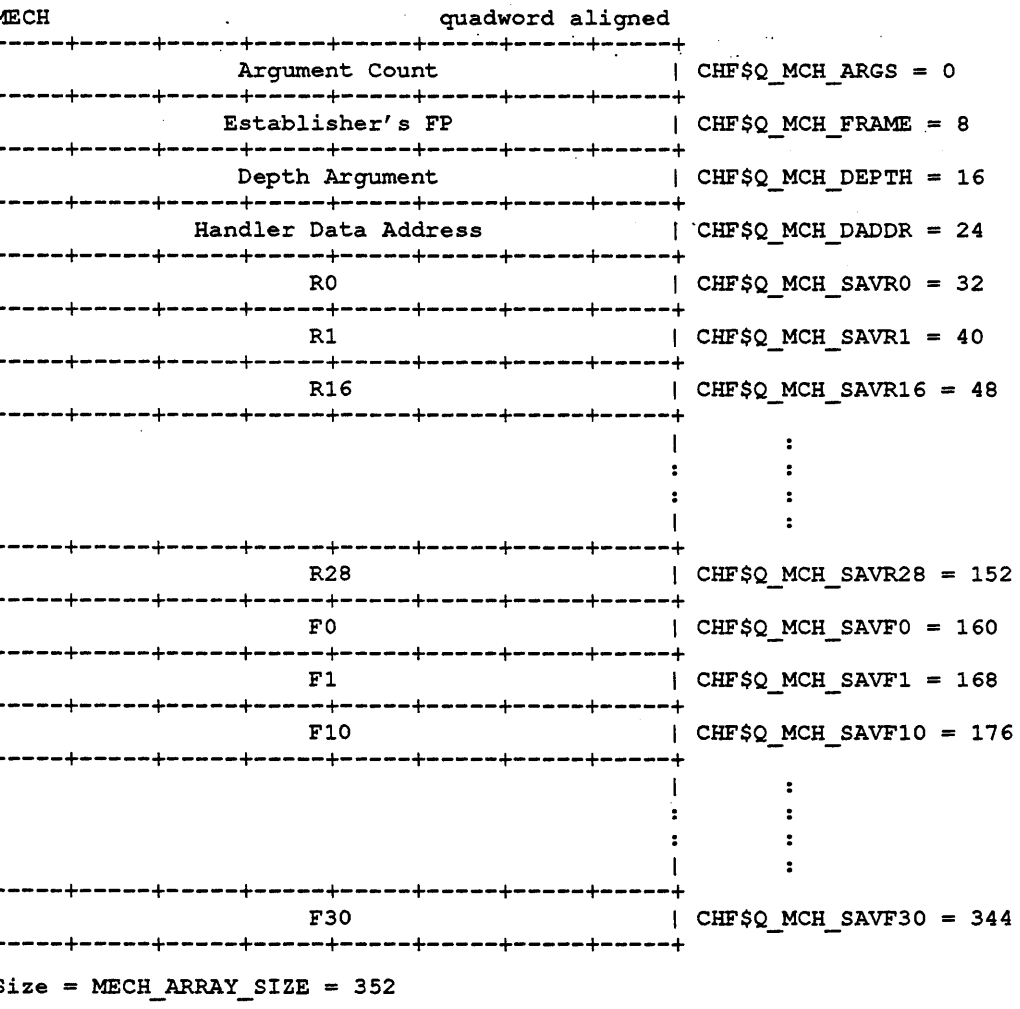
15.1.7 Mechanism Vector

The VAX/VMS mechanism vector includes the fields CHF$L_MCH_SAVR0 and CHF$L_MCH_SAVR1, which refer to registers R0 and R1 respectively. The values of R0 and R1 in the mechanism vector may be changed and then followed by a call to *SYS$UNWIND* which will have the effect of returning the new values as the return value from the unwound procedure.

In the ALPHA-32 environment, the mechanism array is 64 bits wide. CHF$Q_MCH_SAVR0 refers to register R0<63:0>, and CHF$Q_MCH_SAVR1 refers to register R1<63:0>. Access is also provided to F0<63:0> and F1<63:0> as CHF$Q_MCH_SAVF0 and CHF$Q_MCH_SAVF1 respectively so that return values may be modified.

If the exception handler data field is present (as indicated by PDSC.FLAGS.HANDLER_DATA_VALID) then the address of the handler data quadword is passed to the handler in the mechanism array at offset CHF$Q_MCH_DADDR, otherwise this field is passed with contents of zero.

Figure 15–1: Mechanism Array Format



15.1.8 Exception Handling Coexistence

The procedure based exception handling facility defined by this standard can coexist and interoperate with a global POSIX-style Signal Facility.

This section defines the features, and the limits, of such a coexistence.

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

  LIB$SIG_TO_EXPT(..system_defined..)

  This special signal handler will gather the software and/or hardware information associated with a signal and then pass it to the exception handling support code in such a way that it can be processed by normal exception handlers. The exception handling code will then proceed with its normal search and invocation procedures as described in Section 15.1.2, Exception Handling Overview.

If any handler returns with a *continue* indication (see Section 15.1.2), thread execution will resume 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 catchall handler is invoked, with *unpredictable* (system dependent) results.

- When an application which utilizes procedure based exception handling is initialized in an environment where signals are also supported, the run time system will install the special signal handler *LIB$SIG_TO_EXPT()* for each of the signals *SIGFPE*, *SIGSEGV*, *SIGBUS*, *SIGILL*, *SIGEMT*, *SIGIOT*, *SIGSYS*, and *SIGTRAP*.

  Applications and language run time support code may install *LIB$SIG_TO_EXPT()* for other signals if desired.

- If, for any signals for which *LIB$SIG_TO_EXPT()* is installed, a thread directly calls any of: *signal()*, *sigblock()*, *sigsetmask()*, *sigpause()*, *sigvec()*, or directly utilizes the Signal Facility support code in any way not otherwise described by this section, the Signal Facility mechanisms will continue to operate correctly but there will be *undefined* effects on procedure based exception handling of those signals.

  ```
  \\
  Possible effects include disabling of procedure based handling for the affected
  signal(s), by virtue of having modified the action taken for the signal(s).

  It is also possible, in a high-reliability environment, for the Signal Facility
  to be enhanced to permit a higher degree of interoperation with procedure based
  exception handling.  For example, an enhanced signal() routine might record the
  action specified in the direct call to signal(), and take that action if no
  frame-based handlers are located or if all frame-based handlers reraise the
  exception.

  However, this standard does not define the effects of these
  actions by a thread. These effects are system defined and
  therefore undefined for purposes of this standard.
  \\
  ```

- A hardware exception or asynchronous software interrupt may immediately pass control to the Signal Facility, in which case normal exception handlers will not be invoked unless *LIB$SIG_TO_EXPT()* is enabled as a signal handling procedure for the signal being delivered.

  ```
  \\
  This is how ULTRIX is likely to work.
  \\
  ```

  Alternatively, the exception dispatcher may immediately initiate the search for exception handlers, and only pass control to the Signal Facility if all handlers re-raise the exception (i.e. do not continue execution of the thread).

  ```
  \\
  This is the current design of VIP.
  \\
  ```

  Which of these alternatives applies to a given environment is *system defined*, and handlers should not assume one or the other of these alternatives.

15.2 Unwinding

The ALPHA-32 environment exception handling is essentially the same as the conventional VAX/VMS design. That design is defined in Sections "VAX Conditions", "Operations Involving Condition Handlers", "Properties of Condition Handlers", and "Multiple Active Signals" of the *VAX Procedure Calling and Condition Handling Standard, Revision 10.3.*

This basic mechanism has been extended to:

- Provide reinvokable handlers
- Provide exception handlers with a handler specific context

15.2.1 Unwind Initiation

In this flavor of the calling standard an unwind may only be invoked while an exception is *active.*

Initiating a General Unwind

A thread may initiate a general unwind operation by calling a system service. This service is defined as follows:

SYS$UNWIND(..PARAMS..)

See the VAX/VMS documentation for details of usage and parameters of SYS$UNWIND.

Initiating an Exit Unwind

A thread may initiate an exit unwind operation by calling a system library function defined as follows:

LIB$EXIT_UNWIND(STATUS)

Arguments:

STATUS	If non-zero, an unwind status value. If zero, indicates that the default unwind status value should be used.

Function Value:

None.	Unlike *SYS$UNWIND*, this function will never return.

The exit unwind operation terminates the thread after all active procedures have been terminated.

As the procedures in a thread are terminated, they must respond to the exit unwind by performing the appropriate exit actions.

15.2.2 Unwind Completion

In the ALPHA-32 flavor of the calling standard the value that exists in R0 when the unwind completes is the value that is passed implicitly to the unwinder in the mechanism array (see Section 15.1.7, Mechanism Vector). This value should be set appropriately by an exception handler before the unwind is initiated.

15.3 Asynchronous Software Interrupts

Asynchronous software interrupts can arise from a number of system and environment specific mechanisms. Examples of mechanisms which may cause such interrupts are:

- Usage of a POSIX style signal facility
- Usage of a VMS style AST mechanism

Because many of the usage semantics are the same. a given execution environment may chose to provide a connection mechanism between asynchronous software interrupts and the exception handling facility. Such a coexistence is described in Section 15.1.8, Exception Handling Coexistence.

While the existence of asynchronous software interrupts profoundly effects this standard, conventions for their usage, invocation and handling other than mentioned above are beyond the scope of this document.

CHAPTER 16

MULTITHREADED ENVIRONMENT CONVENTIONS

The conventions and considerations for the ALPHA-32 multithreaded environment are the same as for the ALPHA-64 environment with the following caveats:

- In the Thread Environment Block (TEB), although fields are allocated as quadwords their usage may only be of the first longword. This means that code referencing fields such as *errno* must be aware of the true size of the data.
- The names of the routines used to perform various functions will differ (typically by including a '$' character in the name) from the ALPHA-64 flavor of the standard.

```
\\
This is a capability that does not exist yet on either VMS.

Language bindings for library routines, status values, etc. still need to be
agreed upon. The naming conventions used in this version of this standard are
formal, for purposes of specification only, and do not represent the final
language bindings to these interfaces.
\\
```

CHAPTER 17

CODE EXAMPLES

The following example illustrates many of the techniques and conventions mentioned above. This example does *not* presume to represent how a compiler might actually layout memory, or to represent actual ALPHA assembler notation. The code fragments presented do not presume to do anything useful.

```
MODULE X

;       Linkage section for Module X

        .psect      $LINK           ;Read-Only after image activate

X1::                                ;Descriptor for stack frame procedure X1

        .word   <<<PDSC_FLAGS_BASE_REG_IS_FP | PDSC_FLAGS_NATIVE | -
                        PDSC_FLAGS_NO_JACKET>@4> -
                        | PDSC_KIND_FP_STACK > ;flags + kind
        .word   8                   ;Register save area offset
        .long   26                  ;No signature info + Reserved

        .address_q X1_ENTRY         ;Address of entry code sequence for X1
        .long   48                  ;Size of fixed part of stack
        .long   0                   ;Reserved
        .long   <BIT13|BIT14|BIT15|BIT29> ;Integer Register mask
        .long   0                   ;Floating Register mask

Y1_LKP:                             ;Linkage pair for procedure Y1
        .address_q Y1_ENTRY         ;Code address for optimized call
        .address_q Y1               ;Y1's descriptor address(procedure value)

POWER_2_TABLE:                      ;Read-only table of powers of 2
        .long 1,2,4,8,16,32,64,128,256,512,1024,2048,4096,8192

Y_DATA_1_ADDR:                      ;External static variables
        .address_l Y_DATA_1         ;Address of Y_DATA_1
Y_DATA_2_ADDR:
        .address_l Y_DATA_2         ;Address of Y_DATA_2


;       Entry code sequence for procedure X1

        .psect  $CODE               ;Code for Module X

X1_ENTRY::                          ;Address of X1's procedure descriptor is in R27
                                    ; on standard call. Return address is in R26
        LDA     SP,-48(SP)          ;Allocate stack space for X1 stack frame
        STQ     R27,(SP)            ;Set up procedure descriptor pointer
        STQ     R26,08(SP)          ;Save return address
        STQ     R13,16(SP)          ;Save preserved R13
        STQ     R14,24(SP)          ;Save preserved R14
        STQ     R15,32(SP)          ;Save preserved R15
        STQ     FP,40(SP)           ;Save caller's frame pointer
        MOV     SP,FP               ;X1 is now the current procedure

        MOV     R27,R13             ;Move own procedure descriptor address to a
                                    ; preserved register because it is needed
                                    ; after the call
```

```
;       Calculate 2 raised to power of first argument and store in Y_DATA_1

        LDA     R23,POWER_2_TABLE-X1(R13) ;Scratch R23 now has address of
                                          ; powers of 2 table
        SLL     R16,#2,R24        ;Scratch R24 now has offset into table
        ADDL    R23,R24,R23       ;R23 now has address of answer
        LDL     R23,(R23)         ;Fetch answer from memory
        LDL     R24,Y_DATA_1_ADDR-X1(R13) ;R24 now has address of Y_DATA_1
        STL     R23,(R24)         ;Store answer in Y_DATA_1

;       Preserve arguments across call to Y1

        MOV     R16,R14           ;Save arg 1 in preserved R14
        MOV     R17,R15           ;Save arg 2 in preserved R15

;       CALL procedure Y1

        LDQ     R26,Y1_LKP-X1(R13) ;Pick up entry code address
        LDQ     R27,Y1_LKP-X1+8(R13) ;Load his procedure value
        CLR     R25               ;Load AI register
        JSR     R26,(R26)         ;Make call to Y1 leaving return address in R26

;       Store first plus second argument in external Y_DATA_2 and
;       return sum as X1 value

        ADDL    R14,R15,R0        ;Get sum in return value register
        LDL     R23,Y_DATA_2_ADDR-X1(R13)
                                  ;Load scratch R23 with address of Y_DATA_2
        STL     R0,(R23)          ;And store answer in Y_DATA_2 as well

        LDQ     R23,08(FP)        ;Get return address
        LDQ     R13,16(FP)        ;Restore preserved R13
        LDQ     R14,24(FP)        ;Restore preserved R14
        LDQ     R15,32(FP)        ;Restore preserved R15
        LDQ     FP,40(FP)         ;Restore caller's frame pointer. Caller is once
                                  ; again the current procedure.
        LDA     SP,48(SP)         ;Restore caller's SP
        RET     R31,(R23)         ;Return to caller with sum in R0

;------------------------------------------------------------------------------
;------------------------------------------------------------------------------

MODULE Y

        .psect  $LINK             ;Linkage section for Module Y

Y1::                              ;Descriptor for register frame procedure Y1

        .word   <<<PDSC_FLAGS_NATIVE | PDSC_FLAGS_NO_JACKET | -
                  PDSC_FLAGS_HANDLER_VALID>@4> -
                  | PDSC_KIND_FP_REGISTER > ;flags + kind
        .byte   22                ;Caller's FP is saved in R22
        .byte   26                ;Return address is left undisturbed in R26
        .long   26                ;No signature info + Reserved

        .address_q Y1_ENTRY       ;Address of entry code sequence for Y1
        .long   0                 ;Fixed frame Size of Zero
        .long   0                 ;Reserved
        .quad   Y1_HND-.          ;Offset to Y1_HND exception handler

Y1_HND:                           ;Descriptor for exception handler

        .word   <<<PDSC_FLAGS_NATIVE | PDSC_FLAGS_NO_JACKET>@4> -
                  | PDSC_KIND_FP_REGISTER > ;flags + kind
        .byte   23                ;Save caller's FP in R23
        .byte   26                ;Leave return address undisturbed in R26
        .long   26                ;No signature info + Reserved
```

```
        .address_q Y1_HND_ENTRY ;Address of entry code sequence for Y1_HND.
                                ; (Y1_HND_ENTRY is not shown
                                ; in this example.)
        .long   0               ;Fixed frame Size of Zero
        .long   0               ;Reserved

Y_STATIC_DATA:                  ;Address of base of static Y data
        .address_l Y_DATA_BASE

        .psect  $DATA           ;Static data area

Y_DATA_BASE:                    ;Start of static data area

Y_DATA_1::                      ;Global static variable Y_DATA_1
        .long   0               ; (filled in above)
Y_DATA_2::                      ;Global static variable Y_DATA_2
        .long   37


;       Sample code sequence for procedure Y1

        .psect  $CODE           ;Code for Module Y

Y1_ENTRY::                      ;Address of Y1's descriptor is in R27
                                ; on standard call. Return address is in R26
        MOV     FP,R22          ;Save caller's frame pointer in scratch R22
        DRAINT                  ;Force pending exceptions to be raised
                                ; in context of caller
        MOV     R27,FP          ;Y1 is now the current procedure

;       Compute product of global static Y_DATA_1 and Y_DATA_2.
;       Store product in Y_DATA_1.

        LDL     R23,Y_STATIC_DATA-Y1(R27)
                                ;Scratch R23 now has base of data area
        LDL     R24,Y_DATA_1-Y_DATA_BASE(R23) ;Scratch R24 gets Y_DATA_1 value
        LDL     R28,Y_DATA_2-Y_DATA_BASE(R23) ;Scratch R28 gets Y_DATA_2 value
        MULL/V  R24,R28,R24     ;Calculate product
        STL     R24,Y_DATA_1-Y_DATA_BASE(R23) ;Store product in Y_DATA_1
        DRAINT                  ;Synchronize exceptions

;       Return to caller

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

Part III: Appendices

This part of the document describes techniques and details related to various aspects of the ALPHA Calling standard. This part of the document is *not* part of the formal standard. Because of this, *nothing* in this section may be required by conformant software.

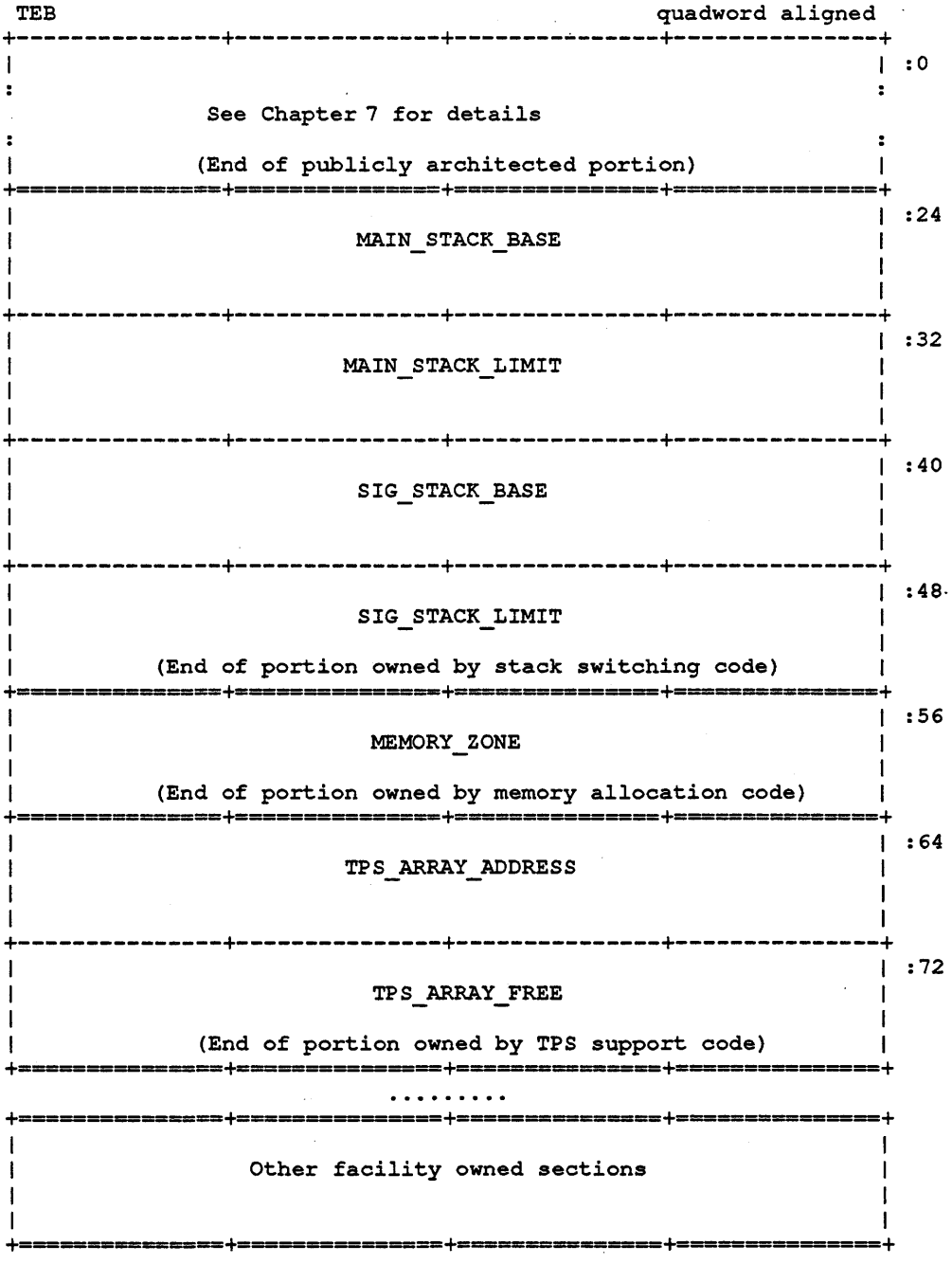
APPENDIX A

MULTITHREAD DESIGN NOTES

This appendix presents a brief overview of some of the implementation details for multithread support. The design reflected here is subject to change without notification. Therefore it is important that compilers and applications *must not* depend on any structures or other information presented in this appendix. Any program or component that depends on this design may not be portable to future implementations of this specification. Portable programs and components must use *only* the functions specified in Chapter 7, Multithreaded Environment Conventions.

A.1 Thread Environment Block Facility Area

Figure A–1: Thread Environment Block Facility Area Format



TEB.MAIN_STACK_BASE contains the highest address allocated to the thread's main stack. The main stack locations in use at any instant are those between the main stack base and the current main stack pointer.

TEB.MAIN_STACK_LIMIT contains the lowest address currently allocated to the thread's main stack. The main stack locations not in use at any time are those between the current main stack pointer – 1 and the main stack limit.

TEB.SIG_STACK_BASE contains the highest address allocated to the thread's signal stack (if any). The signal stack locations in use at any instant are those between the signal stack base and the current signal stack pointer.

The contents of TEB.SIG_STACK_BASE are *unpredictable* except in environments where signal stacks are supported.

TEB.SIG_STACK_LIMIT contains the lowest address currently allocated to the thread's signal stack (if any). The signal stack locations not in use at any time are those between the current signal stack pointer – 1 and the signal stack limit.

The contents of TEB.SIG_STACK_LIMIT are *unpredictable* except in environments where signal stacks are supported.

TEB.MEMORY_ZONE contains the identification of a virtual memory zone which can be used for allocation of heap storage that will be released by the thread support system when the thread terminates.

The run time system uses this field to manage the virtual memory in which the TPS array and all TPS regions for this thread are allocated, and ensures that this storage is released when the thread is terminated.

TEB.TPS_ARRAY_ADDRESS contains a pointer to the data structure that describes each TPS region. When the TEB is created by the system, this field is set to zero, indicating that the TPS array has not been created.

TEB.TPS_ARRAY_FREE contains the byte offset of the first unused element of the TPS array specified by TEB.TPS_ARRAY_ADDRESS. When the TEB is created, this field is set to zero.

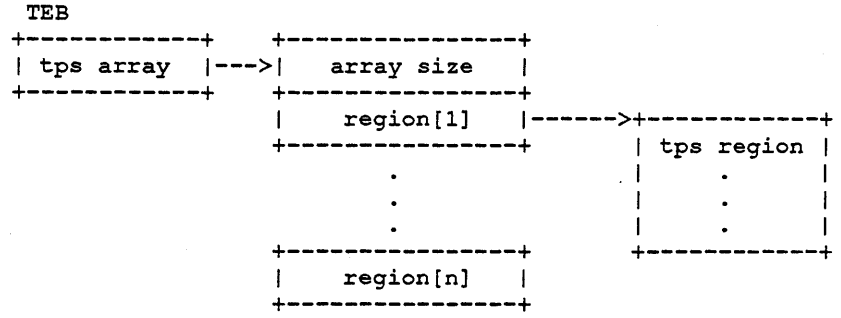
A.2 Design of Thread Private Storage

The use of thread private storage requires support from compilers, the linker, the run time system, and the operating system. This appendix describes only the run time data structures, interfaces, and other conventions of TPS that govern the operation of generated code. (For details of TPS support in the object language, linker processing, TPS initialization, and so forth, consult the *Digital Equipment Corporation ALPHA Object File Format Specification*).

A.2.1 Data Structures for Thread Private Storage

The high level design picture of the run time TPS data structures is as shown here.

Figure A–2: Overview of TPS Data Structures

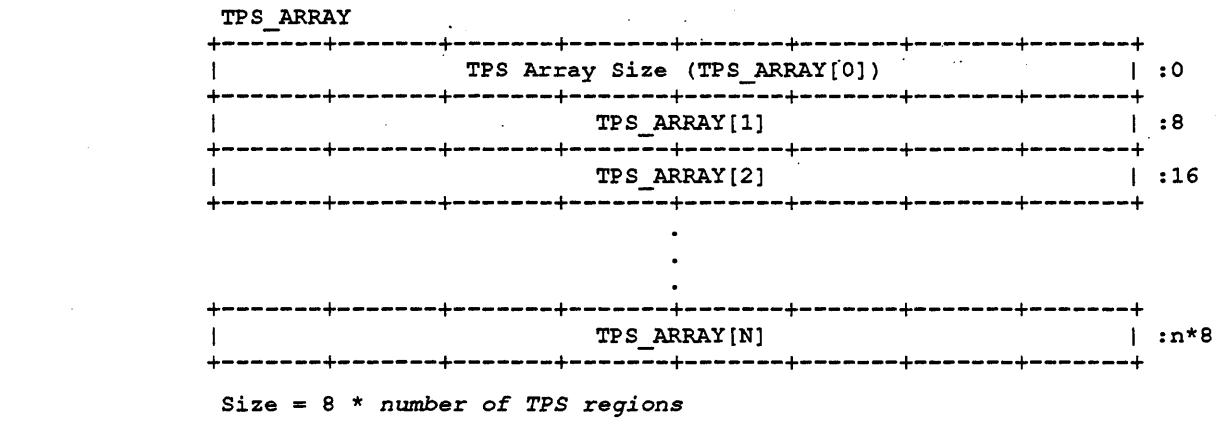


TPS Array

If a thread has access to *n* TPS regions, each of those regions is assigned a unique *array offset* between 8 and *n* * 8. *Array offsets* for programs and runtime code libraries, are assigned by a combination of the linker and the system's image activation code.

Each thread in a process uses the same *array offset* to refer to each instance of a particular TPS region. The *array offset* of a TPS region is used to index a data structure, called the *TPS array*, which holds the base address of the instance of each TPS region for the thread. The TPS array is located at the address contained by TEB.TPS_ARRAY_ADDRESS, and has the format shown here.

Figure A–3: TPS Array Format



TPS_ARRAY[0] specifies the size in bytes of the TPS array, including both used and unused entries.

TPS_ARRAY[i] contains the base address of TPS region *i* for this thread. If *tps_array[i]* is zero, then region *i* has not yet been accessed by this thread, and therefore has not yet been created.

A.2.2 Creating TPS Regions

Non-existent TPS regions must be created before their contents can be accessed. The TPS support code automatically creates the TPS regions the first time a call is made to *LIB_GET_TPS()* for that region. One possible way to accomplish this would be to have a support routine defined as follows:

TPS__CREATE_TPS_REGION(TEB, TPS_DESCRIPTOR)

Arguments:

TEB	The address of the thread environment block.
TPS_DESCRIPTOR	The address of the TPS descriptor for the region.

Function Value:

TPS_REGION_ADDRESS	The base address of the specified TPS region.

The steps performed by *TPS__CREATE_TPS_REGION()* include:

- Creating the thread specific memory zone if it does not exist.
- Creating the TPS array if it does not exist.
- Dynamically extending, and possibly relocating, the TPS array to create a TPS region whose *array offset* is beyond the current extent of the TPS array.
- Maintaining the TEB fields that support the TPS array.
- Maintaining the fields of the TPS array.
- Creating and initializing a TPS region instance if it does not exist.

- Returning the base address of the requested TPS region, whether or not that region already exists before *TPS__CREATE_TPS_REGION()* is called.
- Synchronizing access to the TPS array with asynchronous exception handlers that can also access the TPS array.

If *TPS__CREATE_TPS_REGION()* is called for a TPS region that already exists, the base address of the region is returned with no additional action.

Since the TPS array may have to be expanded and may have to be relocated in order to create a region, any call to *TPS__CREATE_TPS_REGION()* may modify TEB.TPS_ARRAY_ADDRESS and/or TEB.TPS_FREE. However, TPS arrays will never be deallocated by *TPS__CREATE_TPS_REGION()*, and old non-zero values of TEB.TPS_ARRAY_ADDRESS and TEB.TPS_FREE will continue to be valid for the life of the thread.

In addition, because asynchronous exception handlers may be invoked and create TPS regions during the execution of *TPS__CREATE_TPS_REGION()*, additional TPS regions may be created by a call to this function.

A.2.3 Using TPS Regions

Compile Time Referencing of TPS Regions

Compile time access to a TPS region is implemented through the use of a *TPS descriptor* (defined below). These descriptors are constructed from two basic pieces of information that are required to reference a thread private storage region. They are:

1. The *ARRAY_OFFSET* of the TPS region.

   The ARRAY_OFFSET of a TPS region is the *byte offset*, specific to that region, into the TPS array.

   The ARRAY_OFFSET is assigned by a combination of the linker and the system image activator. When any secondary image or runtime code library is loaded into memory, the system develops the offset for each TPS region by adding the offset of the next available slot in the TPS array to the offsets specified in the image being loaded.

   Because of this run time fixup, TPS descriptors are always allocated in a a non-sharable data segment. This is usually the *linkage section*.

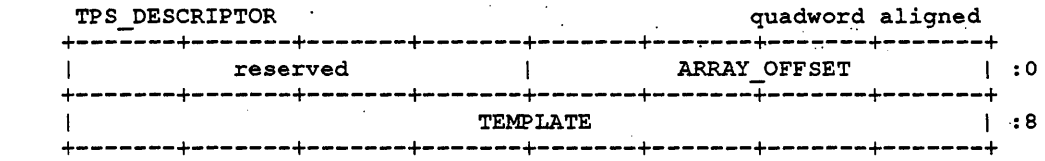
2. The *template* for the TPS region.

   The template for a TPS region is a read-only data structure that describes both the size and initial contents of the TPS region.

   Compilers contribute the size and initial content information for each TPS region. The linker coordinates contributions from multiple modules and determines the size of the TPS region and its initial contents.

A reference to a TPS region actually resolves to the address of a TPS descriptor for that region. The format of a TPS descriptor is shown below:

Figure A–4: TPS Descriptor Format



ARRAY_OFFSET contains the byte offset of the TPS region in the TPS array.

TEMPLATE contains the self-relative offset to the TPS region's allocation and initialization data.

Runtime Accessing of TPS Regions

The Thread Environment Block is the root data structure for accessing a TPS region instance at run time. The TEB contains the following fields to support TPS regions.

- **TEB.MEMORY_ZONE** contains the identification of a virtual memory zone for this thread. When the TEB is created, this field is set to zero, indicating that the zone has not been allocated. The run time system uses this field to manage the virtual memory in which all thread private storage is allocated, and to ensure that this zone is released when the thread is terminated.

  Both the TPS array and TPS regions for this thread are allocated in the thread private virtual memory zone.

The base address of a TPS region is fixed for the life of the thread, and therefore can be reused for subsequent accesses to the region.

The values contained by the TEB are not fixed. Because of dynamic library loading, the size of the TPS array might change, which could require relocating the TPS array and changing TEB.TPS_ARRAY_ADDRESS and TEB.TPS_ARRAY_FREE. As a result, these TEB values may change across any external call. However, if TPS array is moved, the old TPS array is not deallocated so that the compiler is free to maintain the TPS array pointer in a register. A procedure holding onto an old TPS array pointer executes correctly, but somewhat slower, since it may require a call to *TPS__CREATE_TPS_REGION()* to obtain a desired region base address from the new TPS array.

Likewise, this code sequence will work correctly if an asynchronous exception occurs during this sequence or during *TPS__CREATE_TPS_REGION()*, even if an asynchronous exception handler causes one or more TPS regions to be created and/or causes the TPS array to be relocated.

APPENDIX B

PROCEDURE DESCRIPTOR ACCESS ROUTINES

A thread can obtain information about any procedure in its own virtual address space by calling system library functions as defined below.

RATIONALE:

By defining a procedure interface to provide read-only access to information in the procedure descriptors we allow these structures to be extended in the future without breaking (then) existing code.

Get Handler

LIB_GET_PROC_HANDLER(PROC_VALUE)

Arguments:

PROC_VALUE	Any procedure value, either bound or static, for a procedure currently mapped in the thread's virtual address space If zero, indicates data should be returned for the currently executing procedure (i.e. the immediate caller of this library routine)

Function Value:

HANDLER	The procedure value for the handler of the requested routine If the return value is PROC_VALUE_NULL, then no handler is defined in the procedure descriptor of the requested procedure

Get Handler Data

LIB_GET_PROC_HANDLER_DATA_ADDR(PROC_VALUE)

Arguments:

PROC_VALUE	Any procedure value, either bound or static, for a procedure currently mapped in the thread's virtual address space If zero, indicates data should be returned for the currently executing procedure (i.e. the immediate caller of this library routine)

Function Value:

HANDLER_DATA_ADDRESS	The address of the handler data for the handler of the requested routine If zero, indicates that no handler data is defined in the procedure descriptor of the requested procedure

Get Entry Address

A procedure's entry address is defined by this standard to always be located at *procedure_value* + 8 (PDSC.ENTRY). This is true for either bound or static procedure values. This value may be retrieved from higher level languages by calling the procedure below:

LIB_GET_ENTRY_ADDRESS(PROC_VALUE)

Arguments:

PROC_VALUE	Any procedure value, either bound or static, for a procedure currently mapped in the thread's virtual address space If zero, indicates data should be returned for the currently executing procedure (i.e. the immediate caller of this library routine)

Function Value:

ADDRESS	The entry address of the requested routine If the return value is 0, then the procedure value passed to this routine was in error.

Get Procedure Kind

LIB_GET_PROC_KIND(PROC_VALUE)

Arguments:

PROC_VALUE	Any procedure value, either bound or static, for a procedure currently mapped in the thread's virtual address space If zero, indicates data should be returned for the currently executing procedure (i.e. the immediate caller of this library routine)

Function Value:

KIND	The procedure kind of the requested routine

Get Return Register

LIB_GET_RETURN_REG(PROC_VALUE)

Arguments:

PROC_VALUE	Any procedure value, either bound or static, for a procedure currently mapped in the thread's virtual address space If zero, indicates data should be returned for the currently executing procedure (i.e. the immediate caller of this library routine)

Function Value:

REGISTER — The number of the register (0-29) in which the requested procedure saves it's return address

If -1, indicates that the return address for the requested procedure is saved on the stack

Get RSA Offset

LIB_GET_RSA_OFFSET(PROC_VALUE)

Arguments:

PROC_VALUE — Any procedure value, either bound or static, for a procedure currently mapped in the thread's virtual address space

If zero, indicates data should be returned for the currently executing procedure (i.e. the immediate caller of this library routine)

Function Value:

RSA_OFFSET — The offset, from the frame base register, at which the Register Save Area starts

If -1, indicates that the requested procedure does not have a register save area on the stack (It is a register frame procedure).

APPENDIX C

PC MAPPING LOOKUP DESIGN NOTES

This section describes some of the considerations for PC mapped procedure location as might be used to support the ALPHA-64 flavor of this standard.

C.1 Problem Statement

PC mapping techniques are one aspect of a reliable mechanism to allow a call chain to be navigated from the currently executing procedure backwards to the first or topmost procedure in a program. To be able to perform this call chain tracing requires that for a given register set (starting with the current one), there must be a way to interpret the contents of the registers so as to find the caller of the routine for which the register set was constructed. One of the critical pieces of information needed to accomplish this is the return address. The calling standard specifies a structure (the procedure descriptor) that describes a procedure, including, amongst other things, the location where the return address is stored during the execution of the procedure. Thus the problem at hand reduces to one of "given a PC value, either a current PC or a PC that is represented by a return address, find the procedure descriptor that describes the registers represented by that PC". Once the procedure descriptor is found all of the needed information is available to properly interpret the register set.

C.2 Requirements

The ALPHA architecture is designed to support extremely high end implementations. Much consideration has been given to aspects of instruction caches, memory prefetching, branch prediction and other techniques that help optimize code execution. One such technique involves placing seldom used instruction sequences far away from normal code flow so as to maximize the use of instruction caches. This technique, coupled with various other techniques for optimizing code, may cause the code for any given procedure to be manifest as a number of disjoint segments in memory. Any scheme that is utilized to find the proper procedure descriptor must support such disjoint code segments.

C.3 A solution

The PC mapped solution to locating the procedure descriptor utilizes a collection of lookup tables such that for any given PC value one of the following is *always* true:

- Either the PC value exists in a table such that given that PC value the address of the procedure descriptor may be found in a well defined place (such as at the same offset into a table of procedure descriptor addresses).

  or

- Some well defined rule exists which either defines the information needed or defines a method to get to the required information.

An example of such a rule might be "If the PC being looked up is not in the PC lookup tables then the return address can always be found in R26 and that address may be used as a lookup value to find a procedure descriptor that provides the information needed to correctly interpret the current register set". A rule such as this might be adopted to allow short segments of auto-load or environment setup code to not appear in the PC mapping tables.

C.4 Data Structures

The basic information needed to support the solution presented above is a set of triples containing PC ranges and pointers. These triples would contain:

- Low address of PC range
- High address of PC range
- Pointer to procedure descriptor for PC range

Since code may exist in relocatable shared images a tradeoff must be made between image activation overhead and run time interpretation overhead. An example of such a tradeoff might be that the tables defined above may consist of low and high PC ranges relative to the starting address of the containing image rather than absolute address. Such a design obviously reduces image activation time at the expense of the relatively infrequent run time access to the tables.

C.5 Building the Maps

Given that the tradeoffs mentioned above have been considered there is a strong goal to make the PC value to procedure descriptor address translation as fast as possible. To allow a full range of implementation techniques across various operating system versions the exact details of the PC tables must be isolated from non-system level code. It is assumed that compilers will generate appropriate object language to provide the PC mapping information for each distinct code segment and that the exact details of how that information is represented and accessed in memory will be knowledge that will only be in the system supplied:

- Linker
- Image Activator/loader
- LIB_GET_PROC_VALUE routine (defined elsewhere in this document)

This would allow the table entries to be sorted, hashed or otherwise tailored to optimize access.

APPENDIX D

IMPLEMENTORS NOTES

This appendix contains hints and other possibly useful information for software implementors.

D.1 Notes for Compiler Writers 64

D.1.1 Code Optimizations

It should be noted that this standard describes conventions and rules for *Standard conforming calls and procedures*. In many cases some **significant** code speedup can be realized by interprocedural analysis that determines that a standard call is not required. Examples of optimizations that might be done include:

- Not establishing a context in the called procedure
- Not setting up the AI register
- Passing additional arguments in registers
- Treating different registers as preserved and scratch

D.1.2 Data Passing Mechanism

This calling standard does not dictate which of the defined mechanisms should be used by individual compilers. This is because different languages require different mechanisms. To achieve maximum interoperability of cross-language calls the following recommended guidelines have been established for compiler writers.

Key to below:

Key Word	Explanation
speed	Done for speed reasons
lang	"Required" in some sense by language
compat	Done to be compatible with other languages
value	Passed by value
desc	Passed by descriptor
ref	Passed by reference
ref(out)	Passed by reference if a VAR or OUTPUT parameter

Compatible Set:

- Entire list contains all languages that can interchange by selecting the proper parameter declarations (VAR, *, OUTPUT, IN-OUT, etc.)
- Uppercase part are languages that roughly follows the same rules for default cases (i.e. easy interchange)

Table D–1: Argument Passing Mechanism Choices

Language	What	How Passed	Reason	Compatible Set
ADA	Scalars	value,ref(out)	speed	ADA,PAS,C,for,cob,pl/i
	Strings	desc,ref	lang	ADA,PAS,FOR,COB,PL/I
	Arrays	desc,ref	lang	ADA,PAS,c,FOR,COB,PL/I
	Bit vectors < 65 bits	value,ref(out)	speed	ADA,PAS,C,FOR,COB,PL/I
	Records < 65 bits	value,ref(out)	speed	ADA,PAS,C,for,cob,pl/i
	Records > 64 bits	ref	compat	ADA,PAS,C,FOR,COB,PL/I
PAS	Scalars	value,ref(out)	speed	ADA,PAS,C,for,cob,pl/i
	Strings	desc,ref	lang	ADA,PAS,FOR,COB,PL/I
	Arrays	desc,ref	lang	ADA,PAS,c,FOR,COB,PL/I
	Bit vectors < 65 bits	value,ref(out)	speed	ADA,PAS,C,FOR,COB,PL/I
	Records < 65 bits	value,ref(out)	speed	ADA,PAS,C,for,cob,pl/i
	Records > 64 bits	ref	compat	ADA,PAS,C,FOR,COB,PL/I
C	Scalars	value	speed	ADA,PAS,C,for,cob,pl/i
	Strings	ref	lang	C
	Arrays	ref	lang	ada,pas,C,for,cob,pl/i
	Records < 65 bits	value	speed	ADA,PAS,C,for,cob,pl/i
	Records > 64 bits	ref	compat	ADA,PAS,C,FOR,COB,PL/I
FOR	Scalars	ref	lang	ada,pas,c,FOR,COB,PL/I
	Strings	desc	lang	ada,pas,FOR,cob,pl/i
	Arrays	desc,ref	compat	ADA,PAS,c,FOR,COB,PL/I
	Records < 65 bits	ref	lang	ada,pas,c,FOR,COB,PL/I
	Records > 64 bits	ref	lang	ADA,PAS,C,FOR,COB,PL/I
COB	Scalars	ref	lang	ada,pas,c,FOR,COB,PL/I
	Strings	ref	compat	ada,pas,FOR,COB,PL/I
	Arrays	ref	compat	ADA,PAS,c,FOR,COB,PL/I
	Records < 65 bits	ref	lang	ada,pas,c,FOR,COB,PL/I
	Records > 64 bits	ref	compat	ADA,PAS,C,FOR,COB,PL/I
PL/I	Scalars	ref	lang	ada,pas,c,FOR,COB,PL/I
	Strings	desc,ref	compat	ada,pas,FOR,COB,PL/I
	Arrays	desc,ref	compat	ADA,PAS,c,FOR,COB,PL/I
	Records < 65 bits	ref	lang	ada,pas,c,FOR,COB,PL/I
	Records > 64 bits	ref	compat	ADA,PAS,C,FOR,COB,PL/I

For most languages, strings and arrays are passed either by descriptor or reference depending upon how much is known. Programmers will have to know what syntax for the particular language will cause them to be passed compatibly to other languages.

The above guidelines do not prevent individual compilers from choosing implementation mechanisms to meet market requirements. FORTRAN, for example, may have a mode where strings are passed by pointer together with an additional length hidden argument.

D.1.3 Compiler Switches

```
\\
A future version of this document will say something about switches that
compilers may have.
\\
```

Switch to:

- Select ALPHA preferred argument passing or UNIX style arg passing
- FORTRAN may have switch for f77 style argument passing

D.1.4 Exception Handlers

It might appear that due to the imprecise nature of ALPHA exceptions many DRAINT instructions must be used in standard code to achieve reliable results. After close inspection of the rules put forth in Section 6.1.11 and in-depth review of the exact requirements of the exception handler(s) used to support the language in question, most compilers writers will be able to determine that the majority of the DRAINTs are not required. This can result in a significant performance gain over the straight-forward approach.

D.1.5 Optimized TPS Access

The straightforward sequence for determining the address of a TPS region requires a call to a system library function. If a procedure accesses more than one TPS region, this call must be made for each region.

To reduce this overhead, which typically occurs at each entry to a procedure that accesses TPS, optimized protocols can be used which take advantage of the fact that the base address of a TPS region is fixed for the life of the thread, and therefore can be re-used for subsequent accesses to the region.

An example of such an optimized TPS access protocol is presented here.

1. Each module accessing one or more TPS regions defines an additional module private TPS region termed the *module TPS table.*
2. For each TPS region accessed in the module, the compiler allocates one address slot in the module TPS table and assigns a module-wide *array offset* in the table to each region.
3. On entry, each globally visible procedure in a module calls *LIB_GET_TPS()* to get the address of that module's TPS table. This address must be retained in module-wide register or otherwise made available to non-global procedures within the module by some means that does not involve thread private storage.
4. If the module TPS table has not been initialized, then the global procedure calls *LIB_GET_TPS()* for each TPS region accessed anywhere in the module, and assigns the base address of each region to that region's location in the module TPS table.

5. If the module TPS table has already been initialized, then a global procedure assumes that the base address of all other TPS regions accessed in the module are available in the table; it will never be necessary to call *LIB_GET_TPS()* for these other regions.
6. In that module, procedures which are not global need not check whether the module TPS table has been initialized, because a prior global entry point will have ensured its initialization.
7. The base of each TPS region accessed in the module is obtained from the module's TPS table (which is accessed using the register or other means referenced in step 3) rather than from *LIB_GET_TPS()*.

This protocol reduces the number of calls to *LIB_GET_TPS()* from once per region per procedure, to only once per module boundary crossing.

D.2 Notes for RTL Writers

This section contains notes for RTL writers

D.2.1 Invocation Handles

There is a strong desire to make invocation handles representable within 64 bits so that they might be held in a register and passed as values easily. What is presented here is a brief discussion of this area.

Requirements

An invocation handle must be able to uniquely identify any given invocation in the call chain. It must support both standard and non standard calls.

One Solution

In forming a solution for the above requirements the following was noted:

- An SP value is not sufficient to represent a frame since register frame procedures using no stack must be supported.
- Any procedure that has the same stack value as another procedure must save their caller's Return Address value in a register not used by that other procedure

  The top *n* bits of the SP are always the same (where *n* varies from implementation to implementation but is always greater than 2)

  At the time of a call the value of SP is always octaword aligned (low order 4 bits are zero)

Therefore an invocation handle could be implemented as 64 bits comprised as follows:

1. Bits <62:4> of an SP value shifted left 1 - (59 bits of information)

   or'd with

2. The number that represents the register where the caller's Return Address value was saved (PDSC.SAVE_RA) or 0 depending on the procedure type (0 for stack frame procedures) - (5 bits of information).

D.3 Notes for O.S. Writers

D.3.1 Stack Alignment

This standard requires that stacks be octaword aligned at the time a new procedure is invoked. During the body of a procedure, however, there is no requirement to keep this level of alignment (even though it may be beneficial). This implies that any asynchronous interrupt handlers must properly align the stack before any standard calls are made.

D.3.2 Unwinding

Determining Procedure State

For every valid program counter in a thread, the procedure state represented by that PC can be determined by applying the following:

Procedures with a fixed stack size

- Program counter is in the procedure prologue, at or before the address specified by PDSC.SP_SET.

  The procedure's stack frame has not been allocated.
  The return address is in the register denoted by PDSC.ENTRY_RA.
  The scratch registers, except the register denoted by PDSC.ENTRY_RA, may have been modified.
  The saved registers have not been modified.
  No registers have been saved.

- Program counter is in the procedure prologue, after the address specified by PDSC.SP_SET but before the address specified by PDSC.ENTRY_LENGTH.

  SP points to a stack frame of fixed size specified by PDSC.SIZE.
  The return address is in the register denoted by PDSC.ENTRY_RA.
  The scratch registers, except the register denoted by PDSC.ENTRY_RA, may have been modified.
  The saved registers have not been modified, and may be partially stored in the frame.

- Program counter is not in the procedure prologue.

  Register SP points to a stack frame of size specified by PDSC.SIZE.
  The return address is in the location specified by the procedure descriptor (either PDSC.SAVE_RA or RSA.SAVED_RETURN).
  Any saved registers are stored in the register save area, and may have been modified.

Procedures with a variable stack size and no Stack Return

- Program counter is in the procedure prologue, at or before the address specified by PDSC.SP_SET.

  The procedure's stack frame has not been allocated.
  The return address is in the register denoted by PDSC.ENTRY_RA.
  The scratch registers, except the register denoted by PDSC.ENTRY_RA, may have been modified.
  The saved registers have not been modified.
  No registers have been saved.

- Program counter is in the procedure prologue, after the address specified by PDSC.SP_SET but before the address specified by PDSC.ENTRY_LENGTH.

  SP points to a stack frame of fixed size specified by PDSC.SIZE.
  The return address is in the register denoted by PDSC.ENTRY_RA.
  The scratch registers, except the register denoted by PDSC.ENTRY_RA, may have been modified.
  Any saved registers have not been modified, and may be partially stored in the frame.

- Program counter is not in the procedure prologue, and not at a "RET R31,xxx" instruction followed by the signature instruction

  The register indicated by PDSC.FLAGS.BASE_REG_IS_FP points to the fixed portion of the stack frame, of size specified by PDSC.SIZE.
  The return address is in the location specified by the procedure procedure descriptor (either PDSC.SAVE_RA or RSA.SAVED_RETURN).
  Any saved registers are stored in the frame, and may have been modified.

- Program counter is at a "RET R31,xxx" instruction followed by the signature instruction

  The varying portion of the stack frame has been deallocated.
  SP points to the fixed portion of the stack frame, of size specified by PDSC.SIZE.
  The return address is in the register designated by PDSC.SAVE_RA
  Any saved registers have been restored, and are still stored in the frame.

Procedures with a variable stack size and Stack Return

- Program counter is in the procedure prologue, at or before the address specified by PDSC.SP_SET.

  The procedure's stack frame has not been allocated.
  The return address is in the register denoted by PDSC.ENTRY_RA.
  The scratch registers, except the register denoted by PDSC.ENTRY_RA, may have been modified.
  The saved registers have not been modified.
  No registers have been saved.

- Program counter is in the procedure prologue, after the address specified by PDSC.SP_SET but before the address specified by PDSC.ENTRY_LENGTH.

  SP points to a stack frame of fixed size specified by PDSC.SIZE.
  The return address is in the register denoted by PDSC.ENTRY_RA.

The scratch registers, except the register denoted by PDSC.ENTRY_RA, may have been modified.
Any saved registers have not been modified, and may be partially stored in the frame.

- Program counter is not in the procedure prologue.

  The register indicated by PDSC.FLAGS.BASE_REG_IS_FP points to the fixed portion of the stack frame, of size specified by PDSC.SIZE.
  The return address is in the location specified by the procedure procedure descriptor (either PDSC.SAVE_RA or RSA.SAVED_RETURN).
  The saved registers are stored in the frame, and may have been modified.

Reloading Registers

```
\\
A future version of this document may describe how the unwinder might find
registers in the register save area by: ANDing off high bits above register
wanted. Index into table of 256 entries (1 for each byte mask) up to 8 times or
just BLBS loop, etc.
\\
```

D.4 Notes for Application Level Programming

D.4.1 Mapping descriptors from VAX/VMS to ALPHA-64

The following table indicates which ALPHA-64 descriptor types correspond to the various VAX/VMS descriptors:

Table D–2: Descriptor Mapping from VAX/VMS to ALPHA-64

VAX/VMS Descriptor Type	VAX/VMS Name	ALPHA Descriptor Type
S	Fixed Length (DTYPE =DSC$K_DTYPE_T)	FIXED_TEXT
S	Fixed Length (DTYPE≠DSC$K_DTYPE_T)	Not Supported
D	Dynamic String	DYNAMIC_TEXT
V	Variable Buffer	Not Supported
A	Array	BYTE_ARRAY
P	Procedure Descriptor	Not Supported
PI	Procedure Invocation	Not Supported
J	Label	Not supported
JI	Label Incarnation	Not Supported
SD	Decimal String	Not Supported
NCA	Non Contiguous Array	BYTE_ARRAY
VS	Varying String	VARYING_TEXT
VSA	Varying String Array	BYTE_ARRAY
UBS	Unaligned Bit String	FIXED_BIT
UBA	Unaligned Bit Array	BIT_ARRAY
SB	String with Bounds	BYTE_ARRAY
UBSB	Unaligned Bit String with Bounds	BIT_ARRAY

APPENDIX E

EXTENDED MIPS CALLING STANDARD DIFFERENCES

This appendix provides a brief summary of the differences between the ALPHA calling standard defined in this document and the Extended MIPS calling standard defined in "Digital Equipment Corporation Extensions to the MIPS R-Series Language Level Run-Time Software Architecture", Revision 1.0, 1 July 1989 (referred to as the "Extended MIPS calling standard" below).

The topics presented are categorized into the following areas:

- Differences that result from differences in the underlying hardware architecture.
- Differences where a change to the Extended MIPS calling standard is recommended, because of deficiencies in that architecture or because new understanding and/or information indicates a new design is needed.
- Other substantive differences that result from different goals, implementation strategy or other reasons.
- Differences in areas of very limited scope and with substantially no portability impact.
- Potential differences due to unresolved issues.

It is assumed throughout this discussion that addresses contain 32 bits on MIPS based systems and 64 bits on ALPHA based systems. This difference is pervasive and not mentioned further. Similarly and consequently, field position and/or order is not mentioned at all.

E.1 Hardware Based Differences

- The "frame descriptors" of the Extended MIPS calling standard are similar to the "procedure descriptors" of the ALPHA calling standard. The ALPHA terminology more accurately reflects the intimate role played by the descriptor on the ALPHA architecture (notably that, unlike MIPS, the address of a procedure's descriptor must be passed as part of a call to that procedure.)
- The Extended MIPS calling standard frame descriptors and ALPHA calling standard procedure descriptors are similar in some ways and very different in others because they reflect many characteristics of the underlying hardware (size and number of registers, etc.). Use of these structures is limited primarily to compiler writers and intimately related run-time support where there is little potential for portability of the code.

- The *lib_get_frame_desc* routine in the Extended MIPS calling standard is similar to *LIB_GET_PROC_DESC* defined by the ALPHA calling standard. The names reflect the respective terminology on those systems.
- The Extended MIPS calling standard provides a single frame descriptor structure for all forms of frame. The ALPHA calling standard provides separate procedure descriptor structures for each of the three kinds of procedure. The ALPHA calling standard distinction is intended to promote the packing of these descriptors within the sixteen-bit offset for a linkage section pointer.
- The Extended MIPS calling standard specifies the exact instruction that shall be used to raise a stack overflow exception, while the ALPHA calling standard does not.

E.2 Extended MIPS Change Based Differences

- The Extended MIPS calling standard specifies that the first reference to a shared image must be a call, while no such requirement exists in the ALPHA calling standard. This requirement should be dropped from the Extended MIPS calling standard as well.
- The Extended MIPS calling standard leaves the action upon detection of stack overflow as system defined. The ALPHA calling standard requires that an exception be raised. Raising an exception and allowing application handling and recovery is required for (at least) the Ada language.
- Several of the routines specified by Extended MIPS calling standard fail to accomplish their intended purpose because they fail to work with frames that have zero stack size. These routines include:
  - *lib_get_virtual_fp*
  - *lib_get_frame*
  - *lib_get_prior_frame*

  The ALPHA calling standard design is recommended as a successor.

E.3 Substantive Differences

- The ALPHA calling standard defines descriptors and parameter passing rules for inter-language interoperability, while the Extended MIPS calling standard does not. The additional rules are appropriate for enhanced capabilities and to support a multi-language environment that has become expected on traditional Digital operating systems.
- The ALPHA calling standard defines a descriptor for bound procedures, while the Extended MIPS calling standard does not. The additional rules are appropriate for enhanced interlanguage interoperability.

- The Extended MIPS calling standard routines *lib_add_frame_desc* and *lib_remove_frame_desc* as well as *lib_remove_all_frame_descs* correspond approximately to the ALPHA calling standard routines *LIB_ADD_PC_RANGES*, *LIB_REMOVE_PC_RANGES* and *LIB_REMOVE_PDSC_PC_RANGES*. The different names reflect the different usage of frame versus procedure descriptors as well as the anticipated differences in the underlying implementation and semantics. Operating systems may provide the Extended MIPS calling standard routines as a compatibility extension by layering on the routines defined by the ALPHA calling standard.
- An Extended MIPS calling standard handler indicates resumption at the point of an exception by calling *exc_resume* (which does not return) and has no return value. An ALPHA calling standard handler indicates resumption by returning a specific status value (STATUS_CONTINUE for resumption, STATUS_RESIGNAL to indicate further searching for a handler). We anticipate that ALPHA systems that wish to be downward compatible with extended MIPS platforms may provide an *exc_resume* routine as a system compatibility extension.
- The Extended MIPS calling standard has an argument count register named *$ac* in contrast to the argument information (AI) register used by the ALPHA calling standard. The difference in names reflects both differences in the underlying hardware as well as the extended content and usage of the AI register in the ALPHA calling standard.
- The ALPHA calling standard does not define the usage or interactions of the BSD UNIX concept of a *signal stack*. This abstraction has not been endorsed by POSIX because of its non-portable nature. The Extended MIPS calling standard discussions and specifications in this area do not conflict with the ALPHA calling standard. These specifications include those for *lib_enable_sigstack_use*, *lib_disable_sigstack_use* and related topics.

E.4 Inconsequential Differences

- The Extended MIPS calling standard contains a field in a frame descriptor that points to an array that associates exception handlers with the particular exception or class of exceptions each will handle. The ALPHA calling standard allows only a single handler to be associated with a procedure description with no precondition on the invocation of the handler. The functionality provided by the extended MIPS design can be cleanly layered using the ALPHA calling standard.
- The Extended MIPS calling standard provides a "default handler" that is intermediate between the last chance handlers and the system catchall handler, while the ALPHA calling standard does not. The increased complexity was not required for the ALPHA calling standard and may be provided as a compatible extension if desired.
- An ALPHA calling standard exception record contains a KIND field that classifies the exception value while the Extended MIPS calling standard does not. Similarly, some additional flags are defined. These are convenience fields that are not needed for portable code.

- The Extended MIPS calling standard provides *lib_safe_code* and *lib_unsafe_code* to deal with hardware interactions with run-time generated code. There is no corresponding need on ALPHA systems. Dummy *#defines*, macros and/or entry points can be provided on ALPHA systems as appropriate for portability.

E.5 Potential Differences

The following items are currently open issues in the ALPHA calling standard but have specific resolutions in the Extended MIPS calling standard. When the ALPHA calling standard issues are resolved, these differences will either be eliminated (resolved the same way as on MIPS) or re-classified in one of the other groups above.

- The relationship and interoperability of POSIX 1003.1 signals and general exceptions is not yet determined.
- Various details related to threads and CMA are not yet resolved. This might impact, for example:
  - The existence of *LIB_CREATE_THREAD_ZONE*
  - The public contents of TEBs
  - The existence of an IS_ASYNC flag

APPENDIX F

WAIVERS GRANTED

This section contains a list of all known variances from this standard.

Table F–1: Calling Standard Waivers Granted

Granted To:	Date	Details
No Waivers yet.		

INDEX

A

B

C

D

E

F

G

I

K

L

M

P

R

S

T

U

V

Memorandum

To: Evax Joint Software Team (EJST)
cc: Steve Jenkins
Steve Law

Date: September 18, 1991
From: Ayub Khan, Project Leader
ALPHA Languages and Tools

Subject: Calling Standard Used by ALPHA C.

The C compiler used in porting ULTRIX onto ALPHA platform was based on RISC ULTRIX4.x compiler technology. The current effort to port OSF1 onto ALPHA platform is enhancing the same compiler technology. The intent of this paper is to publish the calling standard used by the ALPHA C compiler.

As mentioned earlier, the ALPHA C compiler is based on compiler technology in existence for current ULTRIX releases. Hence, the calling standard used by ALPHA C is based on extending the current technology. The only differences are the one that are dictated by the underlying hardware. In particular we document the following differences:

- Register r0 contains the return address.
- Register r30 contains the stack pointer.
- Register r31 always has the zero value.
- A float or double parameter is assigned a floating point register.
- Register f31 always has the zero value.

As you might have noticed, the differences are limited to the register usage only, the organization of the call frame does not change.

The object file produced by the ALPHA C compiler is XCOFF and will also be supporting ELF.

For your convenience, we are providing you with chpatars on *"Linkage Conventions"* and *"Object File Format"* from our ***Assembly Language Reference*** manual, which document in detail the calling standard used by the ALPHA C compiler.

Linkage Conventions 7

This chapter gives rules and examples to follow when designing an assembly language program. The chapter concludes with a "learn by doing" technique that you can use if you still have any doubts about how a particular calling sequence should work. This involves writing a skeleton version of your prospective assembly routine using a high level language, and then compiling it with the -S option to generate a human-readable assembly language file. The assembly language file can then be used as the starting point for coding your routine.

7.1 Introduction

When you write assembly language routines, you should follow the same calling conventions that the compilers observe, for two reasons:

- Often your code must interact with compiler-generated code, accepting and returning arguments or accessing shared global data.
- The symbolic debugger gives better assistance in debugging programs using standard calling conventions.

The conventions for the compiler system are a bit more complicated than some, mostly to enhance the speed of each procedure call. Specifically:

- The compilers use the full, general calling sequence only when necessary; where possible, they omit unneeded portions of it. For example, the compilers don't use a register as a frame pointer whenever possible.
- The compilers and debugger observe certain implicit rules rather than communicating via instructions or data at execution time. For example, the debugger looks at information placed in the symbol table by a " `.frame`" directive at compilation time, so that it can tolerate the lack of a register containing a frame pointer at execution time.

7.2 Program Design

This section describes three general areas of concern to the assembly language programmer:

- usable and restricted registers
- stack frame requirements on entering and exiting a routine
- the "shape" of data (scalars, arrays, records, sets) laid out by the various high level languages.

7.2.1 Register Use and Linkage

The main processor has 32 64-bit integer registers. The uses and restrictions of these registers are described in Table 7-1.

The floating point coprocessor has 32 floating point registers. Each register can hold either a single precision (32 bit) or a double precision (64 bit) value. Refer to Table 7-2 for details.

Table 7-1: Integer Registers

Register Name	Software Name (from regdef.h)	Use and Linkage
$0	ra	Contains the return address and used for expression evaluation.
$1 or $at	AT	Reserved for the assembler.
$2..$3	v0..v1	Used for expression evaluations and to hold the integer function results. Also used to pass the static link when calling nested procedures.
$4..$7	a0..a3	Used to pass the first 4 integer type actual arguments. Their values are not preserved across procedure calls.
$8..$15	t0..t7	Temporary registers used for expression evaluations. Their values are not preserved across procedure calls.
$16..$23	s0..s7	Saved registers. Their values must be preserved across procedure calls.
$24..$25	t8..t9	Temporary registers used for expression evaluations. Their values are not preserved across procedure calls.
$26..$27	k0..k1	Reserved for the assembler.
$28 or $gp	gp	Contains the global pointer.
$29	s8	A saved register (like s0-s7).
$30 or $sp	sp	Contains the stack pointer.
$31	zero	Always has the value 0.

Table 7-2: Floating Point Registers

Register Name	Use and Linkage
$f0..$f1	Used to hold floating point type function results ($f0) and complex type function results ($f0 has the real part, $f1 has the imaginary part).
$f2..$f11	Temporary registers, used for expression evaluation, whose values are not preserved across procedure calls.
$f12..$f15	Used to pass the first 4 single or double precision actual arguments, whose values are not preserved across procedure calls.
$f16..$f19	Temporary registers, used for expression evaluations, whose values are not preserved across procedure calls.
$f20..$f30	Saved registers, whose values must be preserved across procedure calls.
$f31	Always has the value 0.0.

7.2.2 The Stack Frame

The compilers classify each routine into one of of the following categories:

- non-leaf routines, that is, routines that call other procedures
- leaf routines, that is, routines that do not themselves execute any procedure calls. Leaf routines are of two types:
  - leaf routines that require stack storage for local variables
  - leaf routines that do not require stack storage for local variables.

You must decide the routine category before determining the calling sequence.

To write a program with proper stack frame usage and debugging capabilities, use the following procedure:

1. Regardless of the type of routine, you should include a `.ent` pseudo-op and an entry label for the procedure. The `.ent` pseudo-op is for use by the debugger, and the entry label is the procedure name. The syntax is:

```
        .ent    procedure_name
procedure_name:
```

2. If you are writing a leaf procedure that does not use the stack, skip to step 3. For leaf a procedure that uses the stack or non-leaf procedures, you must allocate all

the stack space that the routine requires. The syntax to adjust the stack size is:

```
subq    $sp,framesize
```

where *framesize* is the size of frame required; *framesize must* be a multiple of 8. Space must be allocated for:

- local variables
- saved general registers. Space should be allocated only for those registers saved. For non-leaf procedures, you must save $0, which is used in the calls to other procedures from this routine. If you use registers $16-$23, you must also save them.
- saved floating point registers. Space should be allocated only for those registers saved. If you use registers $f20-$f30 you must also save them.
- procedure call argument area. You must allocate the maximum number of bytes for arguments of any procedure that you call from this routine.

Note

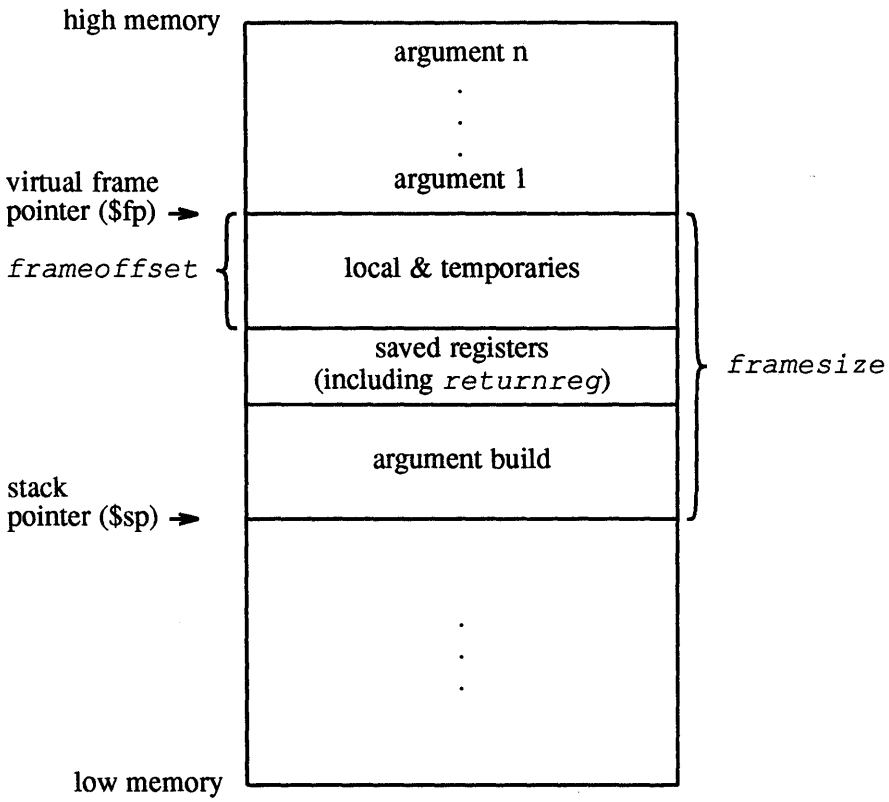
Once you have modified $sp, you should not modify it again for the rest of the routine.

3. Now include a `.frame` pseudo-op:

```
.frame  framereg,framesize,returnreg
```

The virtual frame pointer is a frame pointer as used in other compiler systems but has no register allocated for it. It consists of the *framereg* ($sp, in most cases) added to the *framesize* (see step 2 above). Figure 7-1 illustrates the stack components.

Figure 7-1: Stack Organization



The *returnreg* specifies the register the return address is in (usually $0). These usual values may change if you use a varying stack pointer or are specifying a kernel trap routine.

4. If the procedure is a leaf procedure that does not use the stack, skip to step 7. Otherwise you must save the registers you allocated space for in step 2.

   To save the general registers, use the following operations:

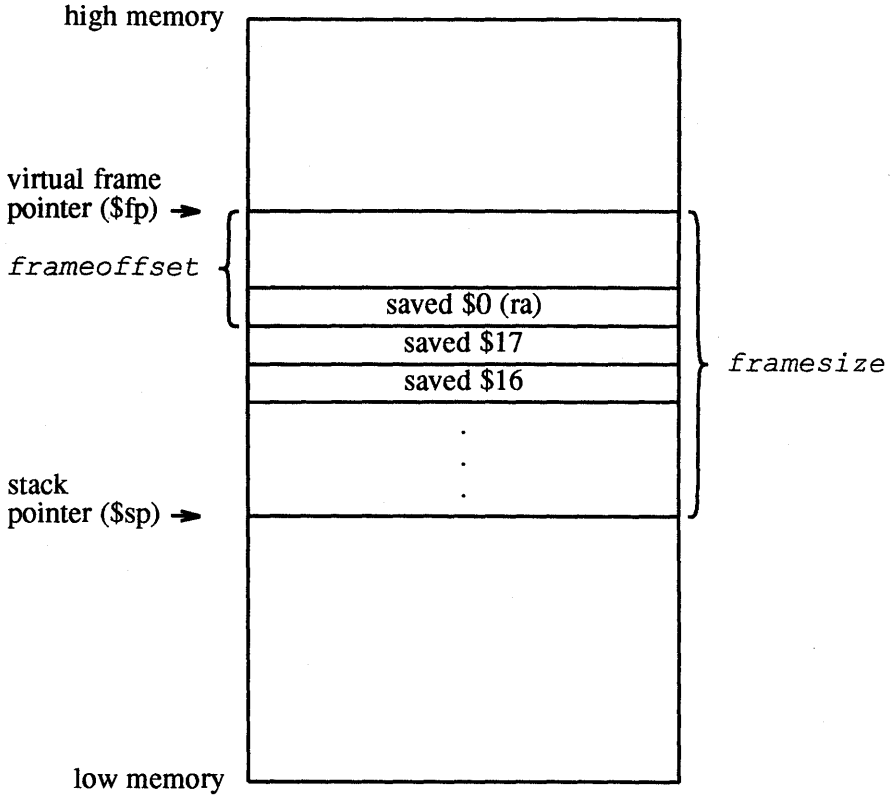
```
.mask   bitmask,frameoffset
stq     reg,framesize+frameoffset-N($sp)
```

   The `.mask` directive specifies the registers to be stored and where they are stored. A bit should be on in *bitmask* for each register saved (for example, if register $0 is saved, bit 0 should be '1' in *bitmask*. The *frameoffset* is the offset from the virtual frame pointer (this number is usually negative). *N* should be 0 for the highest numbered register saved and then incremented by eight for each subsequently lower numbered register saved. For example:

```
stq     $0,framesize+frameoffset($sp)
stq     $17,framesize+frameoffset-8($sp)
stq     $16,framesize+frameoffset-16($sp)
```

   Figure 7-2 illustrates this example.

Figure 7-2: Stack Example



Now save any floating point registers that you allocated space for in step 2 as follows:

```
.fmask  bitmask,frameoffset
st[st]  reg,framesize+frameoffset-N($sp)
```

Notice that saving floating point registers is identical to saving general registers except we use the `.fmask` pseudo-op instead of `.mask` , and the stores are of floating point singles or doubles. The discussion regarding saving general registers applies here as well.

5. This step describes parameter passing: how to access arguments passed into your routine and passing arguments correctly to other procedures. For information on high-level language specific constructs (call-by-name, call-by-value, string or structure passing), refer to Chapter 3 of the *Language Programmer's Guide.*

   As specified in step 2, space must be allocated on the stack for all arguments even though they may be passed in registers. This provides a saving area if their registers are needed for other variables.

   General registers $4-$7 and float registers $f12-$f15 are used for passing the first four arguments. If any of the first four arguments are non-floating point arguments, they are passed in general registers. If any of the first four arguments are floating point arguments, they are passed in float registers if all preceding arguments are floating point; otherwise they are

passed in general registers.

Stack space is used for passing the fifth and subsequent arguments. The stack space allocated to each argument is an 8-byte multiple and is aligned on an 8-byte boundary.

In the table below, the 'f *N*' arguments are considered single and double precision floating point arguments, and 'n *N*' arguments are everything else. The ellipses (...) mean that the rest of the arguments do not go in registers regardless of their type. The register assignments occur in the order shown in order to satisfy optimizing compiler protocols.

Table 7-3: Register Assignments

Arguments	Register Assigments
(f1, f2, f3, f4, ...)	f1 → $f12, f2 → $f13, f3 → $f14, f4 → $f15
(f1, n1, f2, f3, ...)	f1 → $f12, n1 → $5, f2 → $6, f3 → $7
(f1, n1, n2, n3, ...)	f1 → $f12, n1 → $5, n2 → $6, n3 → $7
(n1, n2, n3, n4, ...)	n1 → $4, n2 → $5, n3 → $6, n4 → $7
(n1, f1, n2, n3, ...)	n1 → $4, f1 → $5, n2 → $6, n3 → $7
(n1, f1, f2, f3, ...)	n1 → $4, f1 → $5, f2 → $6, f3 → $7

6. Next, you must restore registers that were saved in step 4. To restore general purpose registers:

```
ldq     reg,framesize+frameoffset-N($sp)
```

   To restore the floating point registers:

```
ld[st]  reg,framesize+frameoffset-N($sp)
```

   (Refer to step 4 for a discussion of the value of *N*).

7. Get the return address:

```
ldq     $0,framesize+frameoffset($sp)
```

8. Clean up the stack:

```
addq    $sp,framesize
```

9. Return:

```
ret     $31,$0
```

10. To end the procedure:

```
.end    procedurename
```

7.2.3 The Shape of Data

In most cases, high-level language routine and assembly routines communicate via simple variables: pointers, integers, booleans, and single- and double-precision real numbers. Describing the details of the various high-level data structures (arrays, records, sets, and so on) is beyond our scope here. If you need to access such a structure as an argument or as a shared global variable, refer to Chapter 3 of the *Language Programmer's Guide*, and the "Learn by Doing" technique described at

the end of this section.

7.3 Examples

This section contains the examples that illustrate program design rules; each example shows a procedure written and C and its equivalent written in assembly language.

Figure 7-3 shows a non-leaf procedure. Notice that it creates a stackframe, and also saves its return address since it must put a new return address into register $0 when it invokes its callee:

Figure 7-3: Non-Leaf Procedure

```
int
nonleaf(i, j)
  int i, *j;
  {
  int abs();
  int temp;

  temp = i - *j;
  return abs(temp);
  }

        .globl  nonleaf
 #    1  int
 #    2  nonleaf(i, j)
 #    3    int i, *j;
 #    4    {
        .ent    nonleaf 2
nonleaf:
        subq    $sp, 56, $sp            ## Create stack frame
        stq     $0, 40($sp)             ## Save the return address
        stq     $4, 56($sp)             ## Save actual arguments
        stq     $5, 64($sp)
        .mask   0x00000001, -16
        .frame  $sp, 56, $0
 #    5    int abs();
 #    6    int temp;
 #    7
 #    8    temp = i - *j;
        ldl     $14, 56($sp)
        ldq     $15, 64($sp)
        ldl     $24, 0($15)
        subl    $14, $24, $25
        stl     $25, 52($sp)
 #    9    return abs(temp);
        bis     $25, $25, $4
        bsr     $0, abs
        ldq     $0, 40($sp)             ## Restore return address
        addq    $sp, 56, $sp            ## Remove stack frame
        ret     $31, $0                 ## Return to caller
        .end    nonleaf
```

Figure 7-4 shows a leaf procedure that does not require stack space for local variables. Notice that it creates no stackframe, and saves no return address:

Figure 7-4: Leaf Procedure Without Stack Space for Local Variables

```
int
leaf(p1, p2)
  int p1, p2;
  {
  return (p1 > p2) ? p1 : p2;
  }

        .globl  leaf
 #   1  leaf(p1, p2)
 #   2    int p1, p2;
 #   3    {
        .ent    leaf 2
leaf:
        .frame  $sp, 0, $0
        addl    $4, 0, $4               ## Truncate p1 to 'int'
        addl    $5, 0, $5               ## Truncate p2 to 'int'
 #   4    return (p1 > p2) ? p1 : p2;
        cmple   $4, $5, $14
        bne     $14, $32
        bis     $4, $4, $6
        br      $31, $33
$32:
        bis     $5, $5, $6
$33:
        bis     $6, $6, $2              ## Return value goes in $2
        ret     $31, $0                 ## Return to caller
        .end    leaf
```

Figure 7-5 shows a leaf procedure that requires stack space for local variables. Notice that it creates a stack frame, but does not save a return address.

Figure 7-5: Leaf Procedure With Stack Space for Local Variables

```
char
leaf_storage(i)
  int i;
  {
  char a[16];
  int j;
  for (j = 0; j < 10; j++)
    a[j] = '0' + j;
  return a[i];
  }

        .globl  leaf_storage
 #   1  char
 #   2  leaf_storage(i)
 #   3    int i;
 #   4    {
        .ent    leaf_storage 2
leaf_storage:
        subq    $sp, 24, $sp            ## Create stack frame
        .frame  $sp, 24, $0
        addl    $4, 0, $4               ## Truncate i to 'int'
 #   5    char a[16];
 #   6    int j;
 #   7    for (j = 0; j < 10; j++)
        stl     $31, 4($sp)
$32:
 #   8      a[j] = '0' + j;
        ldl     $14, 4($sp)
```

```
        addl    $14, 48, $15
        addq    $14, $sp, $24
        stb     $15, 8($24)
        ldl     $25, 4($sp)
        addl    $25, 1, $8
        stl     $8, 4($sp)
        cmplt   $8, 10, $9
        bne     $9, $32
#   9     return a[i];
        addq    $4, $sp, $2
        ldb     $2, 8($2)               ## Return value goes in $2
        addq    $sp, 24, $sp            ## Remove stack frame
        ret     $31, $0                 ## Return to caller
        .end    leaf_storage
```

7.4 Learning by Doing

The rules and parameter requirements required between assembly language and other languages are varied and complex. The simplest approach to coding an interface between an assembly routine and a routine written in a high-level language is to do the following:

- Use the high-level language to write a skeletal version of the routine that you plan to code in assembly language.
- Compile the program using the -S option, which creates an assembly language (.s) version of the compiled source file.
- Study the assembly-language listing and then, imitating the rules and conventions used by the compiler, write your assembly language code.

The next two sections illustrate techniques to use in creating an interface between assembly language and high-level language routines. The examples shown are merely to illustrate what to look for in creating your interface. Details such as register numbers will vary according to the number, order, and data types of the arguments. You should write and compile realistic examples of your own code in writing your particular interface.

7.4.1 Calling a High-Level Language Routine

The following steps show a technique to follow in writing an assembly language routine that calls `atof` a routine written in C that converts ASCII characters to numbers; for more information, see `atof`(3) in the *Alpha/OSF Reference Pages, Section 3*.

1. Write a C program that calls `atof`. Pass global rather than local variables; this makes them easy to recognize in the assembly language version of the C program. (and ensures that optimization doesn't remove any of the code on the grounds that it has no effect.)

   Below is an example of a C program that calls `atof`.

```
char c[] = "3.1415";
double d, atof();
float f;
caller()
  {
  d = atof(c);
  f = (float)atof(c);
  }
```

2. Compile the program using the using the compiler options shown below:

```
cc -S -O caller.c
```

The -S option causes the compiler to produce the assembly-language listing; the -O option, though not required, reduces the amount of code generated, making the listing easier to read.

3. After compilation, look at the file caller.s (shown below). The comments in the listing show how the parameters are passed, the execution of the call, and how the returned values are retrieved.

```
        .globl  c
        .sdata
        .align  2
        .align  0
c:
        .word   875638323 : 1
        .word   13617 : 1
        .comm   d 8
        .comm   f 4
        .text
        .align  2
        .globl  caller
#   1   char c[] = "3.1415";
#   2   double d, atof();
#   3   float f;
#   4   caller()
#   5     {
        .ent    caller 2
caller:
        subq    $sp, 64, $sp
        stq     $0, 48($sp)
        stt     $f20, 40($sp)
        .mask   0x00000001, -16
        .fmask  0x00100000, -24
        .frame  $sp, 64, $0
#   6     d = atof(c);
        la      $4, c                   ## Load address of c
        bsr     $0, atof                ## Call atof
        stt     $f0, d                  ## Store result in d
#   7     f = (float)atof(c);
        la      $4, c                   ## Load address of c
        bsr     $0, atof                ## Call atof
        cpys    $f0, $f0, $f20          ## Copy double result to temp
        cvtts   $f20, $f2               ## Convert double to float
        sts     $f2, f                  ## Store float result in f
#   8     }
        ldt     $f20, 40($sp)
        ldq     $0, 48($sp)
        addq    $sp, 64, $sp
        ret     $31, $0
        .end    caller
```

7.4.2 Calling an Assembly Language Routine

This section shows a technique to follow in writing an assembly language routine that calls a routine written in a high-level language (Pascal is used in this example).

1. Write a facsimile of the assembly language routine you wish to call. In the body of the routine, write statements that use the same arguments you intend to use in

the final assembly language routine. Copy the arguments to global variables rather than local variables to make it easy for you to read the resulting assembly language listing.

Below is the Pascal facsimile of the assembly language program.

```
type
  str = packed array [1 .. 10] of char;
  subr = 2 .. 5;
var
  global_r: real;
  global_c: subr;
  global_s: str;
  global_b: boolean;
function callee(var r: real; c: subr; s: str): boolean;
  begin
  global_r := r;
  global_c := c;
  global_s := s;
  callee := c = 3;
  end;
```

2. Compile the program using the using the compiler options shown below:

   ```
   pc -S -O callee.p
   ```

   The -S option causes the compiler to produce the assembly-language listing; the -O option, though not required, reduces the amount of code generated, making the listing easier to read.

3. After compilation, look at the file callee.s (shown below). The comments in the listing shows how the parameters are passed, the execution of the call, and how the returned values are retrieved.

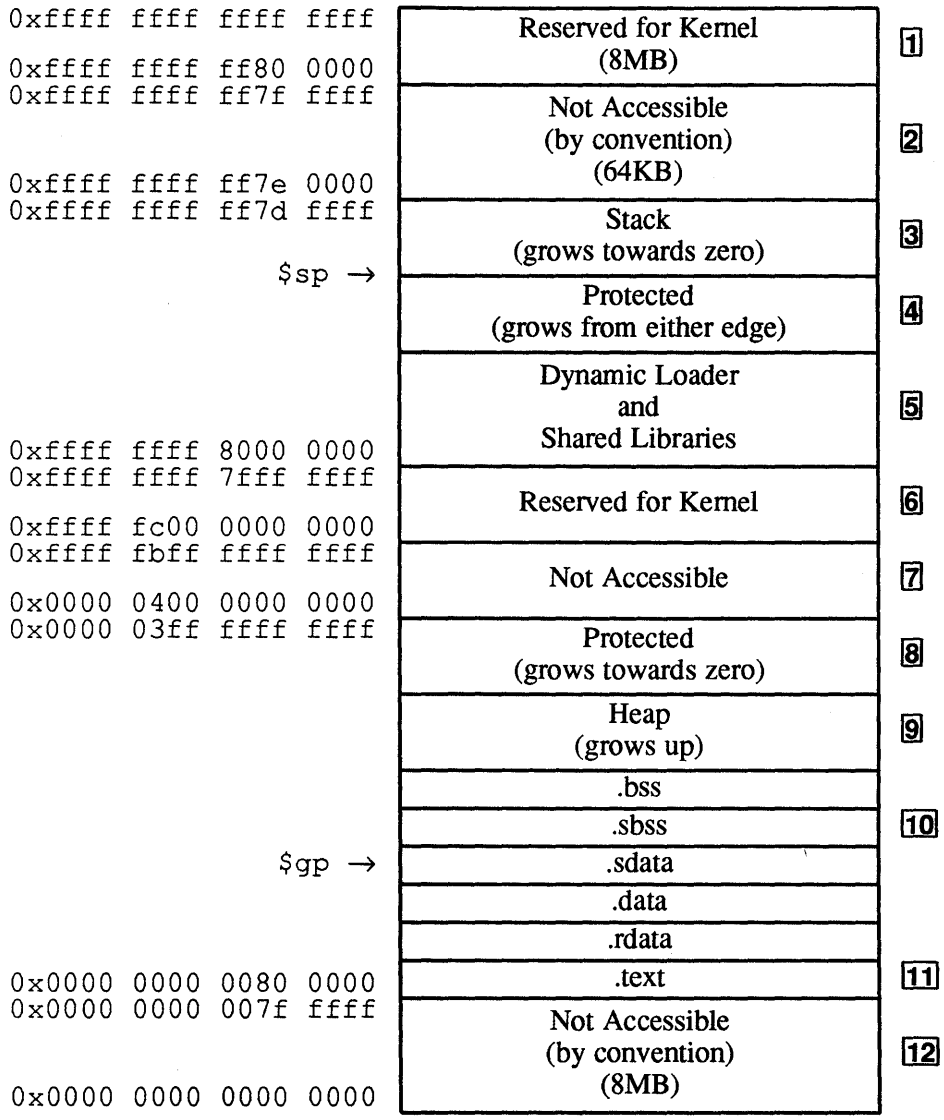
```
        .lcomm  $$65 4
        .comm   global_r 4
        .comm   global_c 1
        .comm   global_s 10
        .comm   global_b 1
        .text
        .align  2
        .globl  callee
 #    9 function callee(var r: real; c: subr; s: str): boolean;
        .ent    callee 2
callee:
        subq    $sp, 8, $sp
        stq     $5, 16($sp)
        stq     $6, 24($sp)
        stq     $7, 32($sp)
        .frame  $sp, 8, $0
 #   10   begin
 #   11   global_r := r;
        lds     $f2, 0($4)      ## The pointer to "r" is in $4
        sts     $f2, global_r
 #   12   global_c := c;
        ldbu    $14, 16($sp)
        stb     $14, global_c
 #   13   global_s := s;
        la      $15, global_s   ## For array "s", the caller copies
        addq    $sp, 24, $24    ## the array to 24($sp)
        li      $8, 10
$32:
        ldbu    $25, 0($24)     ## Copy "s" to global_s using
        stb     $25, 0($15)     ## a tight byte-copying loop.
        addq    $24, 1, $24
```

```
        addq    $15, 1, $15
        subq    $8, 1, $8
        bne     $8, $32
#  14     callee := c = 3;
        ldbu    $9, 16($sp)
        cmpeq   $9, 3, $10
        stb     $10, 7($sp)
#  15     end;
        zapnot  $10, 1, $2      ## Mask the boolean and return in $2
        addq    $sp, 8, $sp
        ret     $31, $0
        .end    callee
```

7.5 Memory Allocation

The machine's default memory allocation scheme gives every process two storage areas, that can grow without bound. A process exceeds virtual storage only when the sum of the two areas exceeds virtual storage space. The link editor and assembler use the scheme shown in Figure 7-6; an explanation of each area in the allocation scheme follows the figure.

Figure 7-6: Layout of memory (User Program View)



1. Reserved for kernel operations
2. Reserved for operating system use.
3. Used for local data in C programs.
4. Not allocated until a user requests it, as in extending the stack or loading a shared library.
5. Reserved for any shared libraries and the dynamic loader.
6. Reserved for kernel operations.
7. Not accessible to kernel or user programs.

8 Not allocated until a user requests it, as in System V shared memory regions.

9 The heap is reserved for sbrk and break system calls, and it not always present.

10 The machine divides all data into one of five sections:

- bss -Uninitialized data with a size greater than the value specified by the -G command line option.
- sbss -Data less than or equal to the -G command line option. (8 is the default value for the -G option.)
- sdata (small data) -Data initialized and specified for the `sdata` section.
- data (data) -Data initialized and specified for the `data` section.
- rdata (read-only data) - Data initialized and specified for the `rdata` section.

11 Contains the .text section

12 Reserved.

Object File Format 9

This chapter provides information on the object file format and has the following major topics:

- An overview of the components that make up the object file, and the differences between the Alpha/OSF object-file format and the UNIX System V common object file format (COFF).
- A description of the headers and sections of the object file. Detailed information is given on the logic followed by the assembler and link editor in handling relocation entries.
- The format of object files (OMAGIC, NMAGIC, and ZMAGIC), and information used by the system loader in loading object files at run-time.
- Archive files and link editor defined symbols.

9.1 Overview

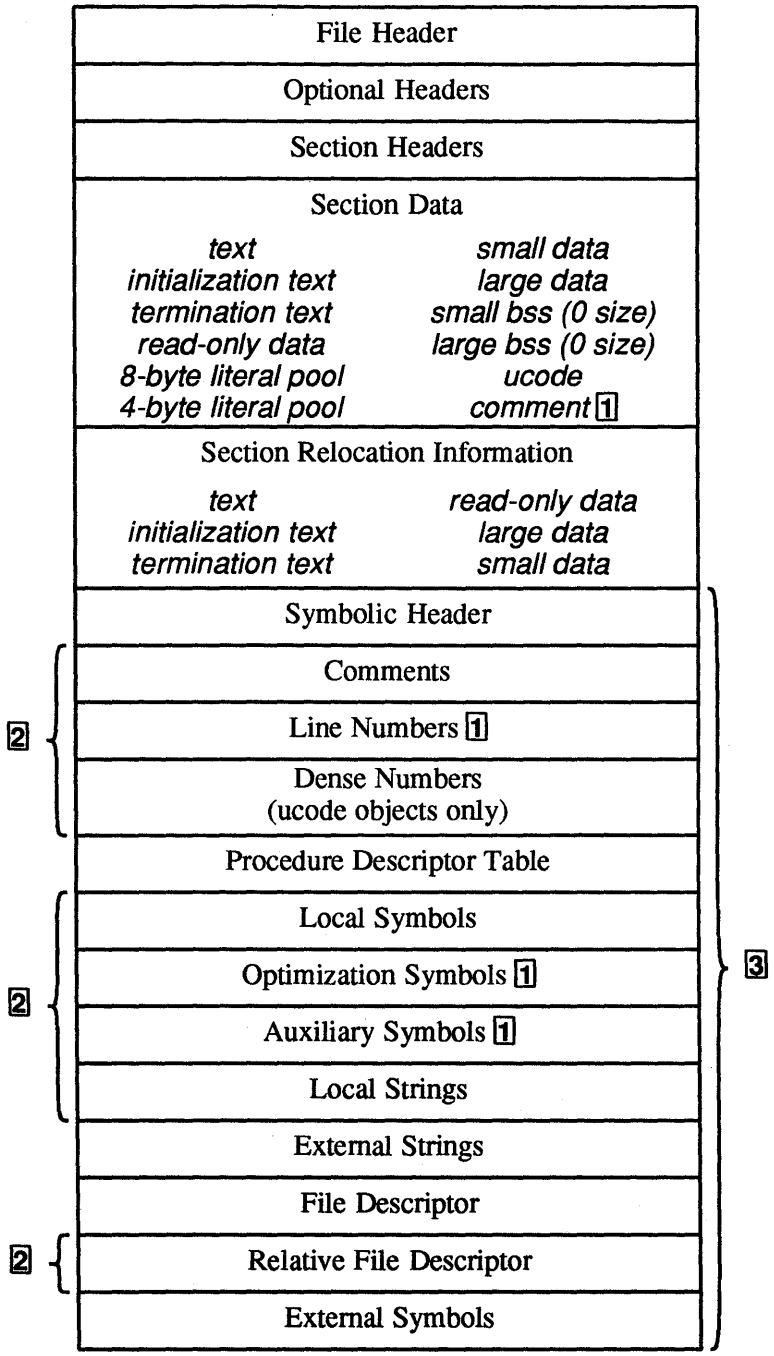
The assembler and the link editor generate object files that have sections ordered as shown in Figure 9-1. Any areas empty of data are omitted, except that the File Header, Optional Header, and Section Header are always present.

The sections of the Symbol table portion (indicated in Figure 9-1) that appear in the final object file format vary, as follows:

- The Line Numbers, Optimization Symbols, and Auxiliary Symbols tables appear only when debugging is on (when the user specifies one of the compiler `-g1`, `-g2` or `-g3` options).
- When the user specifies the `-x` option (strip non-globals) for the link edit phase, the link editor strips the Line Number, Local Symbols, Optimization Symbols, Auxiliary Symbols, Local Strings, and Relative File Descriptor tables from the object file, and updates the Procedure Descriptor table.
- The link editor strips the entire Symbol table from the object file when the user specifies the `-s` option (strip) for the link edit phase.

Any new assembler or link editor designed to work with the compiler system should lay out the object file sections in the order shown in Figure 9-1. The link editor can process object files that are ordered differently, but performance may be degraded.

Figure 9-1: Object File Format



1 Created only if debugging is ON.
2 Missing if stripped of non-globals.
3 Symbol Table. Missing if fully stripped

Readers already familiar with standard UNIX System V COFF (common object file format) may be interested in the difference between it and the compiler system format, as described next.

The compiler system File Header definition is based on UNIX System V header file `filehdr.h` with the following modifications.

- The symbol table file pointer and the number of symbol table entries now specify the file pointer and the size of the Symbolic Header respectively (described in
- All tables that specify symbolic information have their file pointers and number of entries in this Symbolic Header.

The Optional Header definition has the same format as specified in the UNIX System V header file `aouthdr.h`, except the following fields have been added: `bss_start`, `gprmask`, `fprmask`, and `gp_value`. See Table 9-4.

The Section Header definition has the same format as the UNIX System V's header file `scnhdr.h`. except the line number fields are used for global pointers. See Table 9-6.

The relocation information definition is similar to UNIX 4.3 BSD, which has local relocation types; however, you should read the Section Relocation Information section in this chapter for information on differences.

9.2 The File Header

The format of the File Header, defined in `filehdr.h`, is shown in Table 9-1.

Table 9-1: File Header Format

Declaration	Field	Description
unsigned short	f_magic	magic number
unsigned short	f_nscns	number of sections
int	f_timdat	time and date stamp
long	f_symptr	file pointer to symbolic header
int	f_nsyms	size of symbolic header
unsigned short	f_opthdr	size of optional header
unsigned short	f_flags	flags

`f_symptr` points to the Symbolic Header of the Symbol table, and `f_nsyms` gives the size of the header. For a description of the Symbolic Header, see

9.2.1 File Header Magic Field (f_magic)

The magic number in the `f_magic` entry in the File Header specifies the target machine on which an object file can execute. Table 9-2 shows the values and mnemonics for the magic numbers; the header file `filehdr.h` contains the macro definitions.

Table 9-2: File Header Magic Numbers

Symbol	Value	Description
ALPHAMAGIC	0603	Alpha machine code object file
ALPHAUMAGIC	0617	Alpha ucode object file

9.2.2 Flags (f_flags)

The `f_flags` field describes the object file characteristics. Table 9-3 describes the flags and gives their hexadecimal values. The table notes those flags that do not apply to compiler system object files.

Table 9-3: File Header Flags

Symbol	Value	Description
F_RELFLG	0x0001	relocation information stripped from file
F_EXEC	0x0002	file is executable (i.e. no unresolved external references)
F_LNNO	0x0004	line numbers stripped from file
F_LSYMS	0x0008	local symbols stripped from file
F_MINMAL	0x0010	minimal object file (".m") output of fextract[a]
F_UPDATE	0x0020	fully bound update file, output of ogen[a]
F_SWABD	0x0040	file whose bytes were swabbed (in names)[a]
F_AR16WR	0x0080	file has the byte ordering of an AR16WR (e.g. PDP-11/70) machine (it was created there, or was produced by conv)[a]
F_AR32WR	0x0100	file has the byte ordering of an AR32WR machine (e.g. vax)[a]
F_AR32W	0x0200	file has the byte ordering of an AR32W machine (e.g. 3b, maxi, MC68000)[a]
F_PATCH	0x0400	file contains "patch" list in Optional Header[a]
F_NODF	0x0400	(minimal file only) no decision functions for replaced functions[a]
F_MIPS_NO_SHARED	0x1000	cannot be dynamically shared
F_MIPS_SHARABLE	0x2000	a dynamically shared object
F_MIPS_CALL_SHARED	0x3000	dynamic executable
F_MIPS_NO_REORG	0x4000	do not reorder sections
F_MIPS_NO_REMOVE	0x8000	do not reorder sections or remove nops

Table Notes:

a. Not used by compiler system object modules.

9.3 Optional Header

The link editor and the assembler fill in the Optional Header, and the system (kernel) loader (or other program that loads the object module at run-time) uses the information it contains, as described in the section Loading Object Files in this chapter.

Table 9-4 shows the format of the Optional Header (defined in the header file `aouthdr.h`).

Table 9-4: Optional Header Definitions

Declaration	Field	Description
short	magic	See Table 9-5.
short	vstamp	version stamp
long	tsize	text size in bytes, padded to 16-byte boundary
long	dsize	initialized data in bytes, padded to 16-byte boundary
long	bsize	uninitialized data in bytes, padded to 16-byte boundary
long	entry	entry point
long	text_start	base of text used for this file
long	data_start	base of data used for this file
long	bss_start	base of bss used for this file
int	gprmask	general purpose register mask
int	fprmask	floating point register mask
long	gp_value	the gp value used for this object

9.3.1 Optional Header Magic Field (magic)

Table 9-5 shows the values of the `magic` field for the Optional Header; the header file `aouthdr.h` contains the macro definitions.

Table 9-5: Alpha/OSF Magic Numbers

Symbol	Value	Description
OMAGIC	0407	Impure Format. The text is not write-protected or sharable; the data segment is contiguous with the text segment.
NMAGIC	0410	Shared Text. The data segment starts at the next page following the text segment and the text segment is write-protected.
ZMAGIC	0413	The object file is to be demand loaded and has a special format; the text and data segments are separated. Text segment is also write protected. (The Alpha/OSF default). The object may be either dynamic or static.

See the Object Files section in this chapter for information on the format of OMAGIC, NMAGIC, and ZMAGIC files.

9.4 Section Headers

Table 9-6 shows the format of the Section Header (defined in the header file `scnhdr.h`).

Table 9-6: Section Header Format

Declaration	Field	Description
char	s_name[8]	section name
long	s_paddr	physical address
long	s_vaddr	virtual address
long	s_size	section size
long	s_scnptr	file pointer to raw data for section
long	s_relptr	file pointer to relocation
long	s_lnnoptr	file pointer to gp (global pointer) tables
unsigned short	s_nreloc	number of relocation entries
unsigned short	s_nlnno	number of gp tables
int	s_flags	flags

9.4.1 Section Name (s_name)

Table 9-7 shows the defined section names for the `s_name` field of the Section Header; the header file `scnhdr.h` contains the macro definitions.

Table 9-7: Section Header Constants for Section Names

Declaration	Field	Description
_TEXT	".text"	text section
_INIT	".init"	initialization text section for shared libraries
_FINI	".fini"	cleanup text section
_RDATA	".rdata"	read only data section
_DATA	".data"	large data section
_LIT8	".lit8"	8 byte literal pool section
_LIT4	".lit4"	4 byte literal pool section
_SDATA	".sdata"	small data section
_BSS	".bss"	large bss section
_SBSS	".sbss"	small bss section
_UCODE	".ucode"	ucode section
_GOT	".got"	global offset table[a]
_DYNAMIC	".dynamic"	dynamic linking information[a]
_DYNSYM	".dynsym"	dynamic linking symbol table[a]
_REL_DYN	".rel.dyn"	relocation information[a]
_DYNSTR	".dynstr"	dynamic linking strings[a]
_HASH	".hash"	symbol hash table[a]
_MSYM	".msym"	additional dynamic linking symbol table[a]
_CONFLICT	".conflict"	additional dynamic linking information[a]
_REGINFO	".reginfo"	register usage information[a]
_PACKAGE	".package"	package table[a]
_PACKSYM	".packsym"	symbol to package map[a]

Table Notes:

a. These sections exist only in ZMAGIC type files and are used during dynamic linking

9.4.2 Flags (s_flags)

Table 9-8 shows the defined values for `s_flags`; the header file `scnhdr.h` contains the definitions (those flags that are *not* used by compiler system object files are noted).

Table 9-8: Format of s_flags Section Header Entry

Symbol	Value	Description
STYP_REG	0x00	regular section; allocated, relocated, loaded
STYP_DSECT	0x01	dummy; not allocated, relocated, not loaded[a]
STYP_NOLOAD	0x02	noload; allocated, relocated, not loaded[a]
STYP_GROUP	0x04	grouped; formed of input sections[a]
STYP_PAD	0x08	padding; not allocated, not relocated, loaded[a]
STYP_COPY	0x10	copy; for decision function used by field update; not allocated, not relocated, loaded; relocated, and line number entries processed normally[a]
STYP_TEXT	0x20	text only
STYP_DATA	0x40	data only
STYP_BSS	0x80	bss only
STYP_RDATA	0x100	read only data only
STYP_SDATA	0x200	small data only
STYP_SBSS	0x400	contains small bss only
STYP_UCODE	0x800	section contains ucode only
STYP_GOT	0x1000	global offset table[b]
STYP_DYNAMIC	0x2000	dynamic linking information[b]
STYP_DYNSYM	0x4000	dynamic linking symbol table[b]
STYP_REL_DYN	0x8000	dynamic relocation information[b]
STYP_DYNSTR	0x10000	dynamic linking symbol table[b]
STYP_HASH	0x20000	dynamic symbol hash table[b]
STYP_MSYM	0x80000	additional dynamic linking symbol table[b]
STYP_CONFLICT	0x100000	additional dynamic linking information[b]
STYP_REGINFO	0x200000	register usage information[b]
STYP_PACKAGE	0x400000	package table[b]
STYP_PACKSYM	0x800000	symbol to package map[b]
STYP_FINI	0x01000000	.fini section text
STYP_COMMENT	0x02000000	comment section
STYP_LIT8	0x08000000	section 8 byte literals only
STYP_LIT4	0x10000000	section 4 byte literals only
S_NRELOC_OVFL	0x20000000	s_nreloc overflowed, the value is in r_vaddr of the first entry
STYP_LIB	0x40000000	section contains shared library information only
STYP_INIT	0x80000000	section initialization text only

Table Notes:

a. Not used by compiler system object modules.

b. These sections exist only in ZMAGIC type files and are used during dynamic linking

S_NRELOC_OVFL is used when the number of relocation entries in a section overflows the s_nreloc field of the section header. In this case, s_nreloc contains the value 0xffff and the s_flags field has the S_NRELOC_OVFL flag set; the value true is in the r_vaddr field of the first relocation entry for that section. That relocation entry has a type of R_ABS and all other fields are zero, causing it to be ignored under normal circumstances.

Note

For performance reasons, the link editor uses the s_flags entry instead of s_name to determine the type of section. However, the link editor does correctly fill in the s_name entry.

9.4.3 Global Pointer Tables

The gp (global pointer) tables are part of the object file that is produced by the assembler. These are used by the link editor in calculating the best -G *num* to compile the objects are specified as recompilable by the -count option. There is a gp table for the .sdata and .bss sections only.

The gp table gives the section size corresponding to each applicable value specified by the -G *num* option (always including 0), sorted by smallest size first. The s_lnnoptr field in the section header points to this value and the s_nlnno field contains the number of entries (including the header). If there is no *small* section, the related gp table is attached to the corresponding *large* section to provide the link editor with this information.

When an object does not contain a data and bss section, the -G *num* option specified for the object at compilation is unknown. Because the size of the literal pools cannot be known, this complicates the calculation of a best -G *num*. However, a reliable calculation can be made when there is an 8-byte literal pool, which ensures that the object was compiled with a -G of at least eight.

The global pointer table has the following format:

```
union gp_table {
    struct {
        int current_g_value;  /* actual value */
        int unused;
    } header;
    struct {
        int g_value;          /* hypothetical value */
        int bytes;            /* section size corresponding */
                              /* to hypothetical value */
    } entry;
};
```

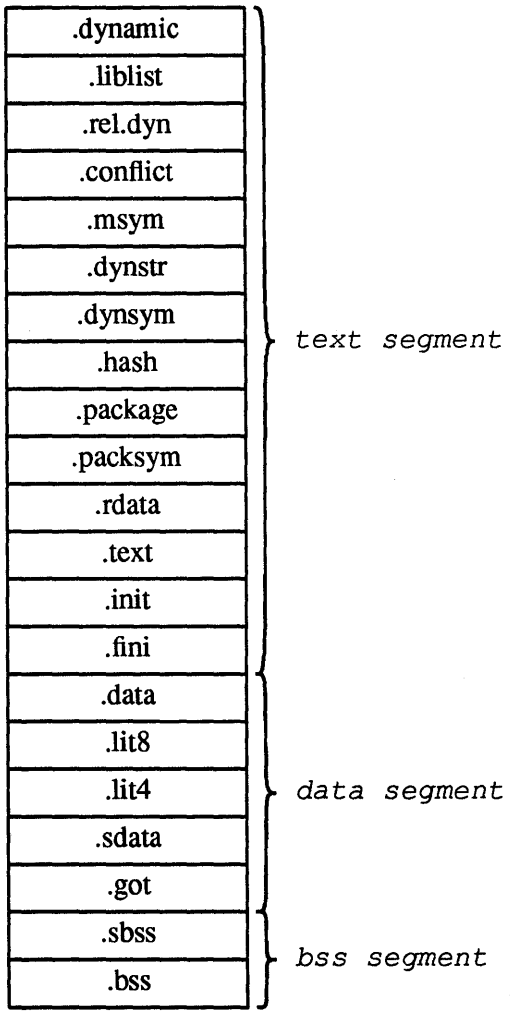
9.5 Section Data

Alpha/OSF compiler system files are represented by the following sections: .dynamic, .liblist, .rel.dyn, .conflict, .msym, .dynstr, .dynsym, .hash, .package, .packsym, .rdata (read-only data), .text, .init (shared library initialization text), .fini (process termination text), .data (data), .lit8 (8-byte literal pool), .lit4 (4-byte literal pool), .sdata (small data), .sbss (small block started by storage), .bss (block started by storage), .lib (shared library information), and .ucode (intermediate code). Figure 9-2 shows the layout of the sections.

The .dynamic, .liblist, .rel.dyn, .conflict, .msym, .dynstr, .dynsym, .hash, .package, and .packsym sections exist only in ZMAGIC files and are used during dynamic linking. These sections are described in more detail in Dynamic linking is discussed in

The .text section contains the machine instructions that are to be executed; the .rdata, .data, .lit8, .lit4, and .sdata contain initialized data, and the .sbss and .bss sections reserve space for uninitialized data that is created by the kernel loader for the program before execution and filled with zeros.

Figure 9-2: Organization of Section Data



As noted in Figure 9-2, the sections are grouped into the text segment (containing the .text, .init, and .fini sections), the data segment (.rdata, .data, .lit8, .lit4, and .sdata), and the bss segment (.sbss and .bss). A section is described by and referenced through the Section Header; the Optional Header provides the same information for segments.

The link editor references the data shown in Figure 9-2 both as sections and segments, through the Section Header and Optional Header respectively. However, the system (kernel) loader, when loading the object file at run-time, references the same data only by segment, through the Optional Header.

9.6 Section Relocation Information

9.6.1 Relocation Table Entry

Table 9-9 shows the format of an entry in the Relocation Table (defined in the header file `reloc.h`).

Table 9-9: Format of a Relocation Table Entry

Declaration	Field	Description
long	r_vaddr	(virtual) address of an item to be relocated.
unsigned	r_symndx:24	index into external symbols or section number; see r_extern below.
unsigned	r_reserved:3	
unsigned	r_type:4	relocation type
unsigned	r_extern:1	= 1 for an external relocation entry; r_symndx is an index into External Symbols. = 0 for a local, relocation entry; r_symndx is the number of the section containing the symbol.

9.6.1.1 Symbol Index (r_symndx) and Extern Field (r_extern)

For external relocation entries, `r_extern` is set to 1 and `r_symnndx` is the index into External Symbols for this entry. In this case, the value of the symbol is used as the value for relocation.

For local relocation entries, `r_extern` is set to 0, and `r_symndx` contains a constant that refers to a section. In this case, the starting address of the section to which the constant refers is used as the value for relocation.

Table 9-10 gives the section numbers for `r_symndx`; the `reloc.h` file contains the macro definitions.

Table 9-10: Section Numbers for Local Relocation Entries

Symbol	Value	Description
R_SN_TEXT	1	.text section
R_SN_INIT	7	.init section
R_SN_RDATA	2	.rdata section
R_SN_DATA	3	.data section
R_SN_SDATA	4	.sdata section
R_SN_SBSS	5	.sbss section
R_SN_BSS	6	.bss section

Table 9-10: (continued)

Symbol	Value	Description
R_SN_INIT	7	.init section
R_SN_LIT8	8	.lit8 section
R_SN_LIT4	9	.lit4 section
R_SN_FINI	12	.fini section

9.6.1.2 Relocation Type (r_type)

Table 9-11 shows valid symbolic entries for the relocation type (`r_type`) field (defined in the header file `reloc.h`).

Table 9-11: Relocation Types

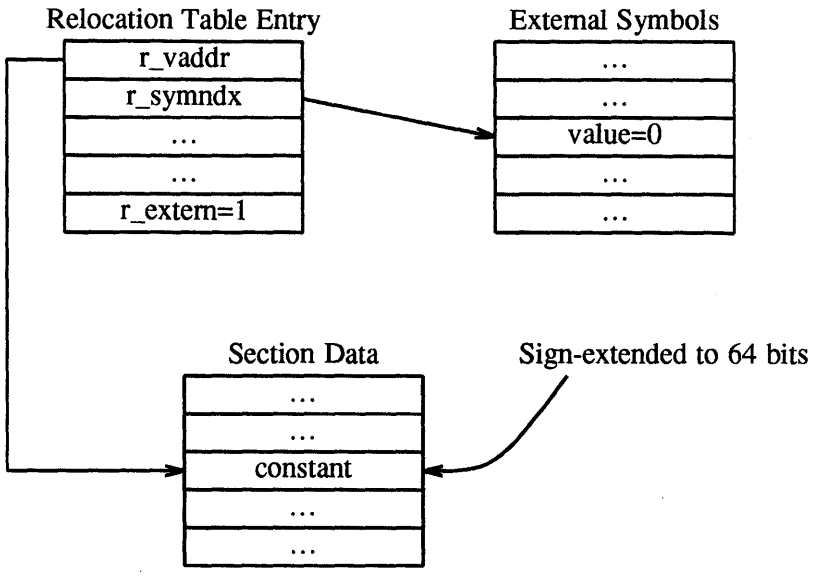
Symbol	Value	Description
R_ABS	0x0	relocation already performed.
R_REFHALF	0x1	16-bit reference to the symbol's virtual address
R_REFWORD	0x2	32-bit reference to the symbol's virtual address
R_JSRADDR	0x3	14-bit jump reference to the symbol's virtual address
R_REFHI	0x4	reference to the high 16-bits of symbol's virtual address
R_REFLO	0x5	reference to the low 16-bits of symbol's virtual address
R_GPREL	0x6	reference to the offset from the global pointer to the symbol's virtual address
R_LITERAL	0x7	reference to a literal in a literal pool as an offset from the global pointer
R_REFDBL	0x8	64-bit reference to the symbol's virtual address
R_BRADDR	0x9	21-bit branch reference to the symbol's virtual address

9.6.2 Assembler and Link Editor Processing

Compiler system executable object modules with all external references defined have the same format as relocatable modules and are executable without re-link editing.

Local relocation entries must be used for symbols that are defined. Therefore, external relocations are used only for undefined symbols. Figure 9-3 gives an overview of the Relocation Table entry for an undefined external symbol.

Figure 9-3: Relocation Table Entry for Undefined External Symbols



The assembler creates this entry as follows:

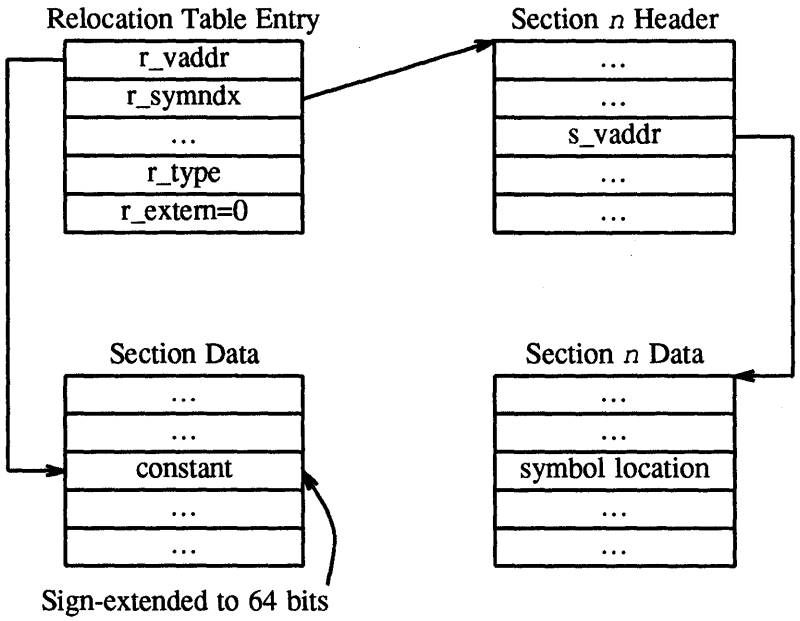
1. Sets `r_vaddr` to point to the item to be relocated.
2. Places a constant to be added to the value for relocation at the address for the item to be relocated (`r_vaddr`).
3. Sets `r_symndx` to the index of the External Symbols entry that contains the symbol value.
4. Sets `r_type` to the constant for the type of relocation types. Table 9-11 shows the valid constants for the relocation type.
5. Sets `r_extern` to 1.

Note

The assembler always sets the value of the undefined entry in External Symbols to 0. It may assign a constant value to be added to the relocated value at the address where the location is to be done. If the width of the constant is less than a full word, and an overflow occurs after relocation, the link editor flags this as an error.

When the link editor determines that an external symbol is defined, it changes the Relocation Table entry for the symbol to a local relocation entry. Figure 9-4 gives an overview of the new entry.

Figure 9-4: Relocation Table Entry for a Local Relocation Entry



To change this entry from an external relocation entry to a local relocation entry, the link editor:

1. Picks up the constant from the address to be relocated (`r_vaddr`).
2. If the width of the constant is less than 64 bits, sign-extends the constant to 64 bits.
3. Adds the value for relocation (the value of the symbol) to the constant and places it back in the address to be relocated.
4. Sets `r_symndx` to the section number that contains the external symbol.
5. Sets `r_extern` to 0.

9.6.2.1 Examples

The examples that follow use external relocation entries.

Example 1: 64-Bit Reference—R_REFDBL. This example shows assembly statements that set the value at location `b` to the global data value `y`.

```
    .globl y
    .data
b:  .quad y # R_REFDBL relocation type at address b for symbol y
```

In processing this statement, the assembler generates a relocation entry of type `R_REFDBL` for the address `b` and the symbol `y`. After determining the address for the symbol `y`, the loader adds the 64-bit address of `y` to the 64-bit value at location `b`, and places the sum in location `b`.

The loader handles 32-bit addresses (R_REFWORD) and 16-bit addresses (R_REFHALF) in the same manner, except it checks for overflow after determining the relocation value.

Example 2: 21-Bit Branch—R_BRADDR. This example shows assembly statements that call routine x from location c.

```
    .text
x:  #routine x
    ...
c:  bsr x # R_BRADDR relocation type at address c for symbol x
```

In processing these statements, the assembler generates a relocation entry of type R_BRADDR for the address and the symbol x. After determining the address for the routine, the loader subtracts the address c+4 to form the displacement to the routine. This result is added to the low 21 bits of the instruction at address c, and after checking for overflow, places the result back into the low 21 bits at address c.

R_BRADDR relocation entries are produced for the assembler's br (branch) and bsr (branch subroutine) instructions.

If the entry is a local relocation type, the target of the branch instruction is assembled in the instruction at the address to be relocated. Otherwise the instruction's displacement field contains a signed offset from the external symbol.

Example 3: High/Low Reference—R_REFHI/R_REFLO. This example shows an assembler macro that loads the absolute address y, plus a constant, into Register 6:

```
    la $6,y+constant
```

In processing this statement, the assembler generates a 0 as the value y, and the following machine language statements:

```
f:  ldah $at,constant>>16($31)    # R_REFHI relocation type at
                                  # address f for symbol y
g:  lda $6,constant&0xffff($at)   # R_REFLO relocation type at address
                                  # g for symbol y
```

In this example, the assembler produces two relocation entries.

Note

When a R_REFHI relocation entry appears, the next relocation entry must always be the corresponding R_REFLO entry. This is required in order to reconstruct the constant that is to be added to the value for relocation.

In determining the final constant values for the two instructions, the link editor must take into account that the lda instruction of the R_REFLO relocation entry sign-extends the immediate value of the constant.

In determining the sum of the address for the symbol y and the constant, the link editor does the following:

1. It uses the low 16 bits of this sum for the immediate value of the R_REFLO relocation address.
2. Because all instructions that are marked with a R_REFLO perform a signed operation, the assembler adjusts the high portion of the sum if Bit 15 is set. Then it uses the high 16 bits of the sum for the immediate value of the R_REFHI instruction at the relocation address.

Example 4: Offset Reference—R_GPREL. This example shows an instruction that loads a global pointer relative value `y` into register 6:

```
ldq $6,y
```

In processing this statement, the assembler generates a 0 as the value `y` and the following machine language statement:

```
h:  ldq $6,0($gp) # R_GPREL relocation type at address h for symbol y
```

and a `R_GPREL` relocation entry would be produced. The assembler then uses the difference between the address for the symbol `y` and the address of the global pointer, as the immediate value for the instruction. The link editor gets the value of the global pointer used by the assembler from `gp_value` in the Optional Header (Table 9-4).

Example 4: Floating Point Literal Reference—R_LITERAL. This example shows an assembler macro that loads a floating point literal into floating point register 0.

```
ldif $f0,1.234
```

is translated into the following machine instruction:

```
h:  lds $f0,-32752(gp)   # R_LITERAL relocation type at
                         # address h for the literal 1.234
```

and a `R_LITERAL` relocation entry is produced; the value of the literal is put into the `.lit4` section. The link editor places only one of all like literal constants in the literal pool. The difference between the virtual address of the literal and the address of the global pointer is used as the immediate value for the instruction. The link editor handles 8-byte literal constants similarly, except it places each unique constant in the `.lit8` section. The value of the `-G` *num* option used when compiling determines if the literal pools are used.

9.7 Object Files

This section describes the object-file formats created by the link editor, namely the Impure (OMAGIC), Shared Text (NMAGIC), and Demand Paged (ZMAGIC) formats. Before reading this section, you should be familiar in the format and contents of the text, data, and bss segments as described in the Section Datasection of this chapter.

The following constraints are imposed on the address at which an object can be loaded and the boundaries of its segments; the operating system can dictate additional constraints.

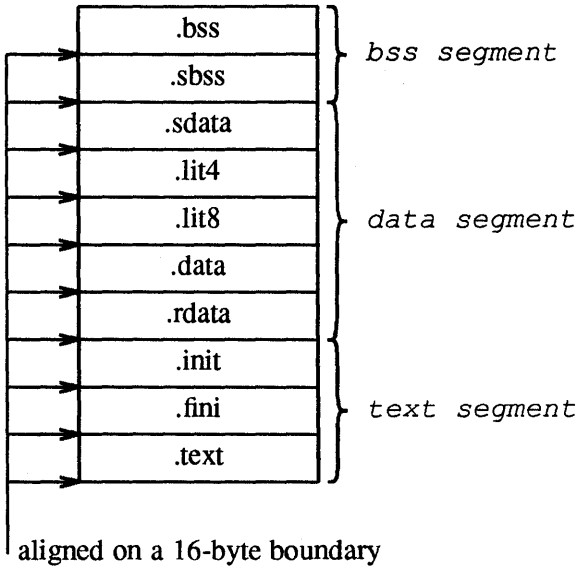
1. Segments must not overlap.
2. Space should be reserved for the stack, which starts below 0xfffffffffff800000 and grows through lower addresses; that is, the value of each subsequent address is less than that of the previous address.
3. The default text segment address for ZMAGIC and NMAGIC files is 0x800000 with the data segment starting at the next 8-megabyte boundary following the end of the text segment.
4. The default text segment address for OMAGIC files is 0x10000000 with the data segment following the text segment.

5. The `-B num` option (specifying a bss segment origin) cannot be specified for OMAGIC files; the default, which specifies that the bss segment follow the data segment, must be used.
6. Alpha/OSF requires a 8-megabyte boundary for segments.

9.7.1 Impure Format (OMAGIC) Files

An OMAGIC file has the format shown in Figure 9-5.

Figure 9-5: Layout of OMAGIC Files in Virtual Memory



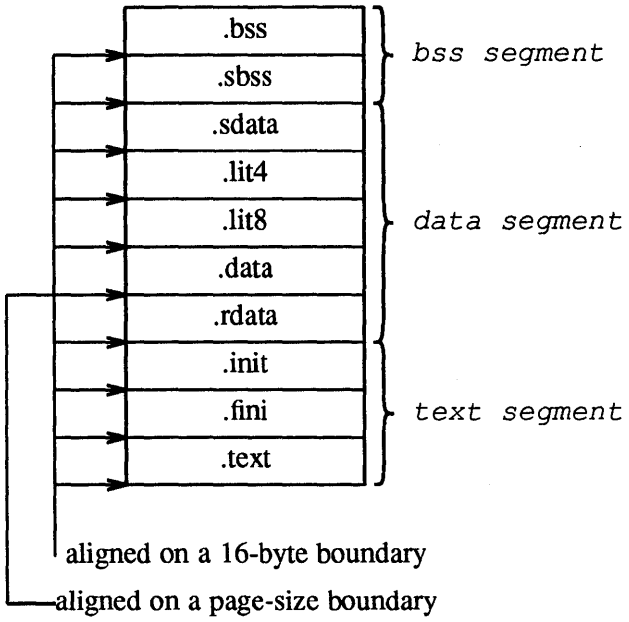
The OMAGIC format has the following characteristics:

- Each section follows the other in virtual address space aligned on an 16-byte boundary.
- No blocking of sections.
- Text, data and bss segments can be placed anywhere in the virtual address space using the link editor's `-T`, `-D` and `-B` options.
- The addresses specified for the segments must be rounded to 16-byte boundaries.
- The text segment contains the `.text`, `.fini`, and `.init` sections.
- The sections in the data segment are ordered as follows: `.rdata`, `.data`, `.lit8`, `.lit4`, and `.sdata`.
- The sections in the bss segment are ordered as follows: `.sbss` and `.bss`.

9.7.2 Shared Text (NMAGIC) Files

An NMAGIC file has the format shown in Figure 9-6.

Figure 9-6: Layout of NMAGIC Files in Virtual Memory



An NMAGIC file has the following characteristics:

- The virtual address of the data segment is on a *pagesize* boundary.
- No blocking of sections.
- Each section follows the other in virtual address space aligned on an 16-byte boundary.
- Only the start of the text and data segments, using the link editor's `-T` and `-D` options, can be specified for a shared text format file; the start of the text and data segments must be a multiple of the *pagesize*.

9.7.3 Demand Paged (ZMAGIC) Files

A ZMAGIC file is a demand paged file in the format shown in Figure 9-7.

A ZMAGIC file has the following characteristics:

- The text segment and the data segment are blocked, with *pagesize* as the blocking factor. Blocking reduces the complexity of paging in the files.
- The size of the sum of the of the File, Optional, and Sections Headers (Table 9-1, Table 9-4, and Table 9-6) rounded to 16 bytes is included in blocking of the text segment.
- The text segment starts by default at 0x800000 (8 Mbyte) , plus the size of the sum of the headers again rounded to 16 bytes. With the standard software, the text segment starts at 0x800000 + header size.

Note

This is required because the first 32K bytes of memory are reserved for future use by the compiler system to allow data access relative to the constant register 0.

- Only the start of the text and data segments, using the link editor's `-T` and `-D` options can be specified for a demand paged format file and must be a multiple of the pagesize.

Figure 9-7: Layout of ZMAGIC Files in Virtual Memory

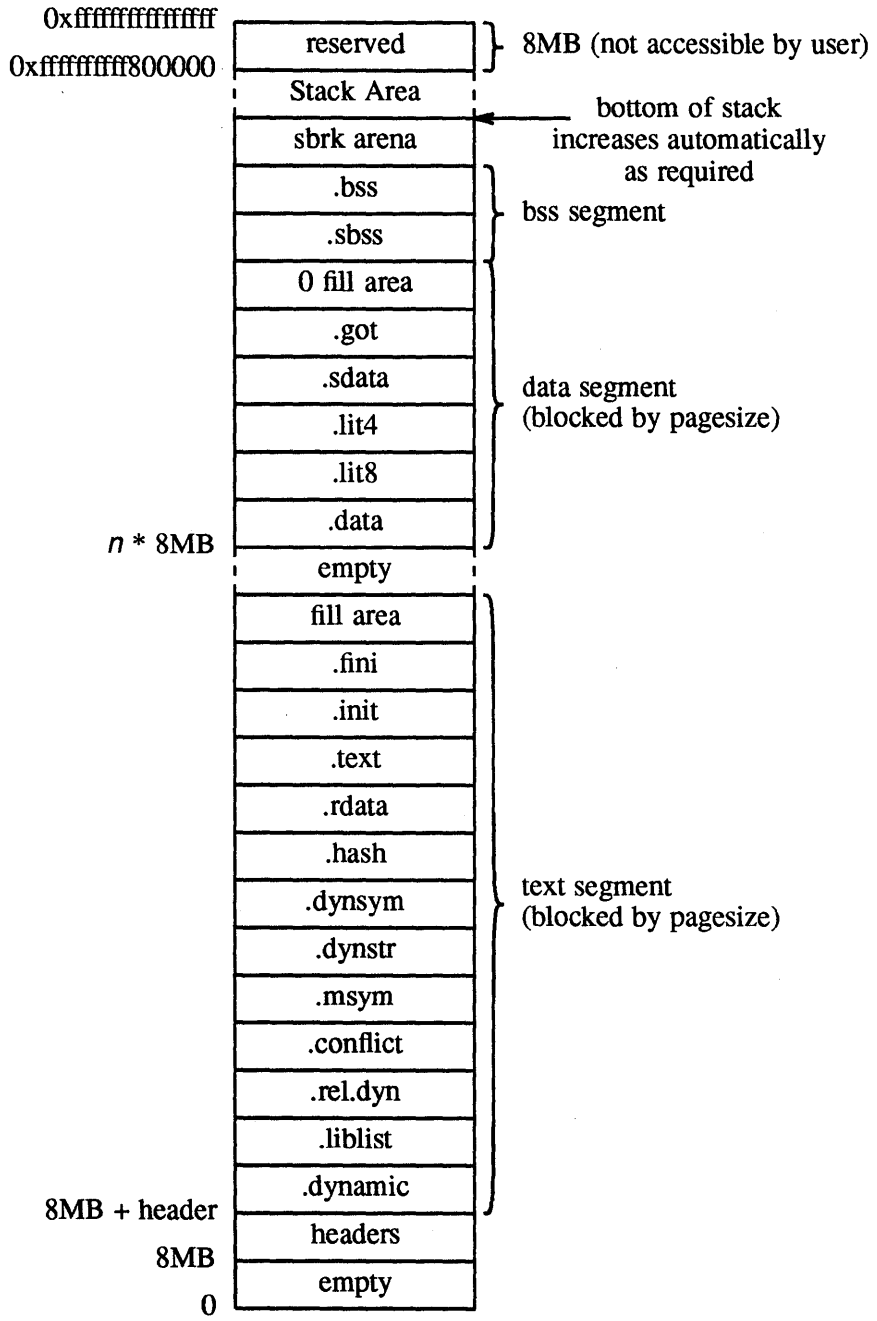
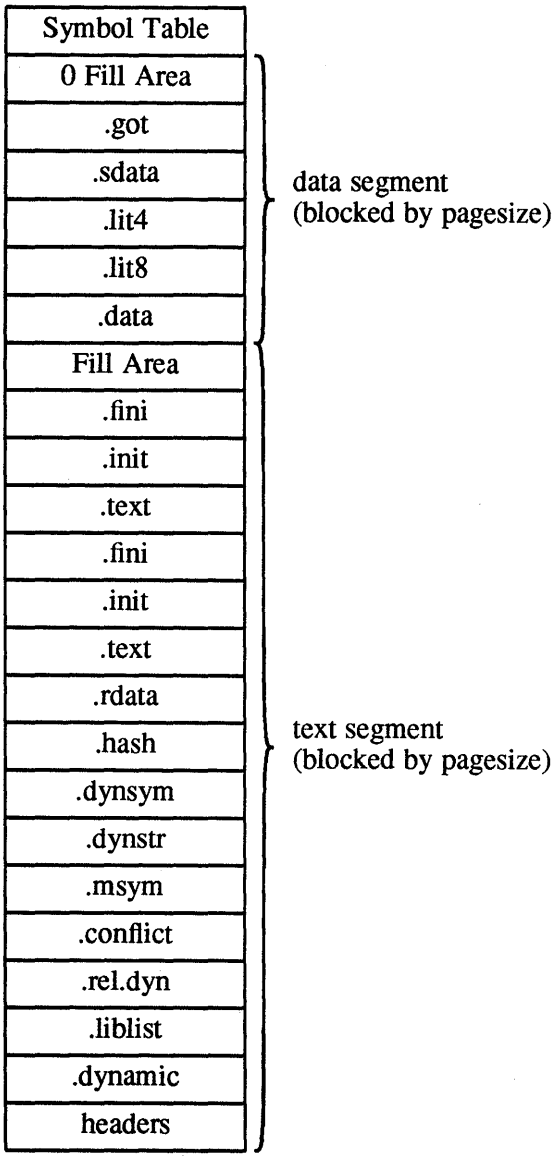


Figure 9-8 shows a ZMAGIC file as it appears in a disk file.

Figure 9-8: Layout of ZMAGIC File on Disk



9.7.4 Ucode objects

Ucode objects contain only a file header, the ucode section header, the ucode section and all of the symbolic information. A ucode section never appears in a machine code object file.

9.7.5 Loading Object Files

The link editor produces object files with their sections in a fixed order similar to UNIX system object files that existed before COFF. See Figure 9.1 for the a description of the sections and how they are formatted.

The sections are grouped into segments, which are described in the Optional Header. In loading the object module at run-time, the system (kernel) loader needs only the magic number in the File Header and the Optional Header to load an object file for execution.

The starting addresses and sizes of the segments for all types of object files are specified similarly, and they are loaded in the same manner.

After reading in the File Header and the Optional Header, the system (kernel) loader must examine the file magic number to determine if the program can be loaded. Then, the system (kernel) loader loads the text and data segments.

The starting offset in the file for the text segment is given by the macro

```
N_TXTOFF(f,a)
```

in the header file `a.out.h`, where *f* is the File Header structure and *a* is the option header structure for the object file to be loaded. The `tsize` field in the Optional Header (Table 9-4) contains the size of the text segment and `text_start` contains the address at which it is to be loaded.

The starting offset of the data segment follows the text segment. The `dsize` field in the Section Header (Table 9-6) contains the size of the data segment; `data_start` contains the address at which it is to be loaded.

The system (kernel) loader must fill the `.bss` segment with zeros. The `bss_start` field in the Optional Header specifies the starting address; `bsize` specifies the number of bytes to be filled with zeros. In ZMAGIC files, the link editor adjusts `bsize` to account for the zero filled area it created in the data segment that is part of of the `.sbss` or `.bss` sections.

If the object file itself does not load the global pointer register it must be set to the `gp_value` field in the Optional Header Table 9-4.

The other fields in the Optional Header are `gprmask` and `fprmask`, whose bits show the registers used in the `.text`, `.init`, and `.fini` sections. They can be used by the operating system, if desired, to avoid save register relocations on context-switch.

9.8 Archive files

The link editor can link object files in archives created by the archiver. The archiver and the format of the archives are based on the UNIX System V portable archive format. To improve performance, the format of the archives symbol table was changed so that it is a hash table, not a linear list.

The archive hash table is accessed through the `ranhashinit()` and `ranlookup()` library routines in `libmld.a`, which are documented in the manual page `ranhash`(3x). The archive format definition is in the header file `ar.h`.

9.9 Link Editor Defined Symbols

Certain symbols are reserved and their values are defined by the link editor. A user program can reference these symbols, but can not define one; an error is generated if a user program attempts to define one of these symbols. Table 9-12 lists the names and values of these symbols; the header file `sym.h` contains their preprocessor macro definitions.

Table 9-12: Link Editor Defined Symbols

Symbol	Value	Description
_ETEXT	"etext"	1st location after .text
_EDATA	"edata"	1st location after .sdata (all initialized data)
_EN	"end"	1st location after .bss (all data)
_FTEXT	"_ftext"	1st location of .text[a]
_FDATA	"_fdata"	1st location of .data[a]
_FBSS	"_fbss"	1st location of the .bss[a]
_PROCEDURE_TABLE	"_procedure_table"	runtime procedure table
_PROCEDURE_TABLE_SIZE	"_procedure_table_size"	runtime procedure table size
_PROCEDURE_STRING_TABLE	"_procedure_string_table"	string table for runtime proc.
_COBOL_MAIN	"_cobol_main"	1st cobol main symbol
_GP	"_gp"	the value of the global pointer[b]

Table Notes:

a. compiler system only.

The dynamic linker also reserves and defines certain symbols; see and for more information.

The first three symbols come from the standard UNIX system link editors and the rest are compiler system specific. The last symbol is used by the start up routine to set the value of the global pointer, as shown in the following assembly language statements:

```
.globl _GP
la $gp,_GP
```

The assembler generates the following machine instructions for these statements:

```
a:  ldah $gp,0($31)   # R_REFHI relocation type at address a for symbol _GP
b:  lda $gp,0($gp)    # R_REFLO relocation type at address b for symbol _GP
```

which would cause the correct value of the global pointer to be loaded.

The link editor symbol _COBOL_MAIN is set to the symbol value of the first external symbol with the `cobol_main` bit set. COBOL objects uses this symbol to determine the the main routine.

9.9.1 Runtime Procedure Table Symbols

The three link editor defined symbols, _PROCEDURE_TABLE, _PROCEDURE_TABLE_SIZE and _PROCEDURE_STRING_TABLE, relate to the runtime procedure table. The Runtime Procedure Table is used by the exception

systems in ADA, PL/I and COBOL. Its description is found in the header file `sym.h`. The table is a subset of the Procedure Descriptor Table portion of the Symbol Table with one additional field, `exception_info`.

When the procedure table entry is for an external procedure, and an External Symbol Table exists, the link editor fills in `exception_info` with the address of the external table. Otherwise, its fill in `exception_info` with zeros.

The name of the External Symbol Table is the procedure name concatenated with the string `_exception_info` (actually, the preprocessor macro EXCEPTION_SUFFIX as defined in the header file `exception.h`.

The Runtime Procedure Table provides enough information to allow a program to unwind its stack. It is typically used by the routines in `libexc.a`. The comments in the header file `exception.h` describes the routines in that library.

The Runtime Procedure Table is sorted by procedure address and always has a dummy entry with a zero address and a 0xffffffff address. When required, the table is padded with an extra zero entry to ensure that the total number of entries is an uneven (odd) number.

The Runtime Procedure Table and String Table for the runtime procedure table are placed at then end of the `.data` section in the object file.