The PRISM Extended Calling Standard

Version 0.7

March 1988

Digital Equipment Corporation

Preface

This document summarizes that part of the PRISM Common Software Architecture that corresponds to the calling standard, and describes the conventions governing the run time environment of PRISM programs.

This document defines the run time data structures, constants, algorithms, conventions, methods, and functional interfaces that enable a a user mode procedure to operate correctly in a multilanguage environment on PRISM systems.

All contents of this document apply to *both* PRISM ULTRIX and MICA. This document specifies the PRISM-specific user mode run time environment which is common and compatible across both operating systems.

Calling standard conventions covered in this document include:

- Register use
- Linkage section
- Invocation descriptors
- Call conventions
- Call stack and frame structure
- Entry and return code sequences
- Exception synchronization and memory synchronization
- Procedure values
- Argument passing mechanisms
- Argument list structure
- Argument descriptors
- Argument data types
- Function value return
- Status codes and condition values
- Rules for FLBC usage and the FLBC displacement field
- Condition handling
- Stack unwinding
- Asynchronous software conditions
- Use of R3 and user mode thread context
- Address values in reserved pages (page zero, page maximum, etc.)
- Stack limit checking
- Alignment rules
- Standard record mappings

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

Those standards, described by other documents, include:

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

Conventions Used in This Document

The definitions in this standard are presented as follows.

- Data Structures

  Data structures are defined in terms of the physical memory layout that must be used for each structure in the PRISM-32 environment.

- Constants

  Constants are presented symbolically. An appendix provides the symbolic to literal translation for each constant.

- 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 precisely, in a form appropriate to each convention.

- Methods

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

- Functional Interfaces

  Functional interfaces are presented in precise abstract form. The semantic capabilities of each functional interface are defined. The language-level interface syntax is not defined.

  For each high level language, each PRISM system must provide an appropriate concrete interface to each abstract functional interface defined by this standard.

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

Revision History

Version 0.1: Initial Proposals, July 1985—R. Grove, M.D. MacLaren

Initial proposals.

Version 0.2: First Distribution, July 1986—C. Nylander

Reflect revision of hardware architecture.
Numerous minor technical revisions and additions.
Cleanup and format for first distribution as Base Document.

Version 0.3: Internal Working Version, November 1986—C. Nylander

Rewrite and reorganize.
Revise in response to comments on first distribution.
Revise scalar register conventions to provide more argument and scratch registers.
Add vector register conventions.
Revise entry descriptor layout and encoding.
Delete dynamic condition handling.
Add reinvokable condition handlers.
Make condition handler specification self-relative.
Revise scalar register saving algorithm.
Elaborate argument passing mechanisms.
Add argument list structure and algorithms.
Add argument datatypes.
Add argument descriptors.
Add function value return.
Revise stack limit checking technique.

Version 0.4: Internal Working Version, December 1986—C. Nylander

Delete standard parent frame pointer (R12), move R22 (argument list address) to R12.
Change RTL_BASE to TEB_BASE.
Add justification for vector register conventions.
Clarify restrictions on lightweight procedures.
Change Entry Descriptor to Frame Descriptor.
Revise frame descriptors to support vector register save/restore.
Revise rules for FP<2:0>.
Revise call frame layout.
Add procedure synchronization rules.
Clarify argument passing mechanisms.
Clarify argument list structure.
Add vector register saving algorithm.
Revise physical format of descriptors for better uniformity and run time efficiency.
Add bit string descriptors.
Define semantics of text string descriptors.
Define function value return by descriptor.
Add sections describing user mode thread architecture.
Add summary of scope and applicability.
Add guidelines section.

Revise usage of "condition" and "exception" to be as consistent with Working Design Document as possible.

Version 0.5: Second Distribution, January 1987—C. Nylander

Expand examples.
Cleanup for second distribution as Extended Calling Standard.

Version 0.6: Internal Working Version, January 1988—C. Nylander

Many miscellaneous clarifications and corrections resulting from review comments.
Minor reorganization.
Add condition handling, signals, and unwinding.
Clarify that entry addresses in invocation descriptors are absolute.
Make R4:R5 destroyable between calling and called routines (e.g. by the auto-loader).
Disallow quadword immediate arguments in standard calls.
Change algorithm for packing procedure arguments in registers.
Disallow descriptors that specify overlapping elements.
Re-do descriptors to unify string and 1 dimensional array descriptors.
Clarify rules for set types in arguments and function value return.
Clarify reference mechanism for returning strings and arrays.
Add new mechanisms for returning function values by descriptor: into fixed buffer, on top of stack, and optionally by dynamic string.
Clarify that pointers are 32 bits.
Add alignment rules.
Add record layout rules.
Remove much of the existing user mode thread architecture section, and rewrite much of what is left.
Change nomenclature around "conditions", "signals", and "exceptions".
Change nomenclature around "Frame Descriptors" (now "Invocation Descriptors") and call frames for heavyweight and lightweight procedure.

Version 0.7: Internal Working Version, March 1988—C. Nylander

Editorial fixups.
Add linkage pairs.
Specify that FP<1:0> can be dirty when FP<2> = 0 as well as when FP<2> = 1.
Generalize quadword immediate arguments to large immediate arguments.
Define trailing null or omitted arguments.
Limit maximum natural alignment requirements to quadword.
Remove process environment block and process control region.
Define reserved memory pages.
Add extra level of indirection to function value return by dynamic string descriptor.
Move function value return by dynamic string descriptor to appendix.
Move thread local storage to appendix.
Move thread local context to appendix.
Change literal constants to symbolic constants.
Make functional interfaces abstract.
Make EXTENT undefined for arrays with more than one dimensions.
Specify that procedure invocations are removed from the invocation chain as unwinding proceeds.
Delete condition records.

Reorganize condition vectors.
Rename condition vectors to be condition records.
Redesign mechanism record—remove all CONDITION fields and most ESTABLISHER fields.
Add abstract interfaces for fetching active procedure context removed from the mechanism record.
Extend invalid stack handling to condition stack.
Define abstract functional interfaces for raising conditions, establishing vectored handlers, establishing alternate condition stack, and unwinding.
Add nested unwinds.
Add overlapping unwinds (informal description)

1 Introduction

This standard defines the rules and conventions that govern the user mode run time environment on PRISM systems. It specifically applies to both PRISM ULTRIX and PRISM MICA systems.

This standard defines properties of the run time environment that must apply at various points during program execution. 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.

Not all of these properties have the same scope. Some properties apply at all points throughout the execution of user mode code, and must therefore be held constant at all times; such properties include those defined for the stack pointer and the frame pointer. Other properties apply only at certain points, such as call conventions that apply only at the point that a JSR to a called procedure is executed.

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

Section "Scope and Applicability" summarizes the points at which elements of this standard apply and the circumstances under which they apply.

This standard defines the *software implementation architecture* for PRISM systems. The conventions described in this standard by definition differ from other software implementation architectures, and programs that depend on properties of this architecture may not be portable to other architectures.

Since source level compatibility and portability between VAX and PRISM is an explicit goal, users should not depend on the properties of this architecture except indirectly through high level language facilities that are portable across architectures.

1.1 Goals

The PRISM calling standard has many of the same goals as the VAX calling standard.

- The standard must be applicable to all intermodule callable interfaces in the PRISM software system. Specifically, the standard must consider the requirements of important compiled languages including ADA, BASIC, BLISS, C, COBOL, FORTRAN, PASCAL, PILLAR, 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 reuse of PRISM software modules are specific goals.

- The standard should provide the programmer with control over fixing, reporting, and flow of control when software conditions or hardware exceptions 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 condition handlers and should minimize time overhead for establishing handlers at the cost of increased time overhead when exceptions occur.

New goals for the PRISM calling standard include:

- Provide a common, compatible user mode run time environment on both the PRISM ULTRIX and PRISM MICA operating systems.
- Maintain high level language source level compatibility with VAX procedure calls. In particular, provide immediate value, reference and descriptor mechanisms for passing arguments.
- Provide a compatible calling standard architecture for a future 64-bit extension of the PRISM architecture.
- Effectively use a large number of registers.
- Pass some arguments in registers to improve performance.
- Provide an efficient mechanism for calling lightweight procedures that do not need a stack call frame because they do not modify preserved registers.
- Use the same calling sequence to invoke lightweight procedures that maintain only a register call frame and and heavyweight procedures that maintain a stack call frame. From the caller's point of view, this unifies the VAX concepts of JSB and CALL linkages. The compiler determines whether to use a stack frame based on the complexity of the called procedure, but this does not require any recompilation of callers.
- Provide condition handling, traceback, and debugging for lightweight procedures that do not have a stack frame.

2 Register Usage Conventions

2.1 Scalar Register Conventions

R0	Hardware defined: binary zero as a source operand, sink (no effect) as a result operand.
R1	SP, the stack pointer. Used by hardware. Must be quadword aligned.
R2	FP, the frame pointer. Defined by call stack conventions. Must be quadword aligned
R3	TEB_BASE, the thread environment block pointer. Contains the address of a data structure used by compiled code and the run time library to maintain thread local context.
R4..R5	Scratch registers which may be modified after the caller has executed the JSR to effect the call, but before the called procedure is invoked. That is, R4..R5 are destroyable between calling and called procedures (e.g. by the auto-loader), before they can be accessed by the called procedure.
R8..R9	In a standard call that returns a function result in a register, the result is returned in R8 (result size $\leq$ 32 bits), or R8..R9 (33 bits $\leq$ result size $\leq$ 64 bits).
R10	In a standard call, R10 contains the address of the called procedure's invocation descriptor.
R11	In a standard call, R11 contains the return address.
R12	In a standard call to a procedure with more than eight longwords in the argument list, R12 contains the quadword aligned memory address of the remainder of the argument list.
R13	In a standard call, R13 contains the number of longwords in the argument list.
R14..R21	In a standard call, the first eight longwords of the argument list are passed in R14..R21.
R4..R31	Scratch registers in standard call, which may be modified by the called procedure without being saved and restored.
R32..R63	No conventional use. If a standard-conforming procedure modifies one of these registers, it must save and restore it.

2.2 Vector Register Conventions

V0..V15	Scratch registers in standard call.
VL	Scratch register in standard call.
VC	Scratch register in standard call.
VM	Scratch register in standard call.

The PRISM calling standard specifies no conventions for preserved vector registers, vector argument registers, or vector function value return registers. All such conventions are by agreement between the calling and called procedures.

Although no vector registers are preserved in a standard call, the entry descriptors defined by this standard include specification of the vector registers saved by a procedure. This so that, when vector registers are preserved by agreement between the calling and called procedure, the condition handling facility will restore them on unwind.

Any condition handler that disturbs the vector state must save and restore it.

\\\\
These conventions for vector register use, particularly the absence of preserved registers, reflect our current best judgement. The rationale for these decisions includes the following assumptions:

1. Nearly all vector arguments will be passed between cooperating procedures, such as math library routines and compiler generated code, that will establish interface-specific agreements for vector register use.

2. Nearly all use of vector operations will be in bottom level procedures (procedures that do not call other procedures), and these procedures should have as many scratch registers as possible available to them without the necessity of saving and restoring vector registers.
3. In those cases where vector registers are used in a calling procedure and there are no agreements with the called procedure, values in the vector registers either do not need to be saved across calls or else can be saved equally well by the calling procedure.

These conventions are subject to change when we have more experience with the use of vector registers in calling and called procedures.
\\

3 Invocation Descriptors

An *Invocation Descriptor* is a quadword aligned data structure that provides basic information about a procedure. This data structure is used in calls between procedures and in interpreting the call stack that exists at any point in a thread's execution.

Some PRISM procedures allocate call frames on the stack, others maintain their call frame entirely in registers (although they may use the stack), and very simple procedures do not necessarily allocate any stack storage at all. The calling procedure need not distinguish these cases. The invocation descriptor for the current procedure contains a field that indicates whether the procedure allocates a call frame on the stack.

As illustrated in the subsequent subsections, the form of an invocation descriptor depends on whether or not a procedure requires a call frame on the stack.

3.1 Invocation Descriptor—Procedures With a Register Frame

A procedure which does not allocate a call frame on the stack (sometimes referred to as a "lightweight procedure") saves it's invocation context in registers.

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

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

```
                                                   Quadword Aligned
+---------------+---------------+---------------+---------------+
|                          CODE_ADDRESS                          | :0
|                           (32 bits)                            |
+---------------+---------------+---------------+---------------+
|   RA_SAVE     |   SP_SAVE     |    FP_SAVE    |  FLAGS  | FMT | :4
|   (8 bits)    |   (8 bits)    |    (8 bits)   |   (5)   | (3) |
+---------------+---------------+---------------+---------------+
|                       CONDITION_HANDLER                        | :8
|                           (32 bits)                            |
+---------------+---------------+---------------+---------------+
```

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

FMT = INVDESCR$C_REGISTER_FRAME for an invocation descriptor denoting a procedure with a register frame.

FLAGS is a set of flag bits defined as follows.

- FLAGS<HAS_HANDLER> is 1 if this procedure has a condition handler. If FLAGS<HAS_HANDLER> is 1, then the CONDITION_HANDLER field must denote a condition handler.
- FLAGS<HANDLER_REINVOKABLE> is 1 if the condition handler is a reinvokable handler, that is, one which can be reinvoked should another condition occur while the handler is already active. If FLAGS<HANDLER_REINVOKABLE> is 0 then the condition handler is not reinvokable.

FLAGS<HANDLER_REINVOKABLE> must be 0 unless FLAGS<HAS_HANDLER> is 1.

- FLAGS<2:4> must be 000.

FP_SAVE is the number of the register in which the value of FP at entry to this procedure is maintained.

SP_SAVE is the number of the register in which the value of SP at entry to this procedure is maintained. SP_SAVE will specify R1 if this procedure does not modify SP.

RA_SAVE 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 R11, then RA_SAVE will specify R11.

CONDITION_HANDLER is a signed self-relative pointer to the invocation descriptor for a run time static condition handling procedure. This field is *not* defined unless FLAGS specifies that there is a condition handler.

```
\\
Lightweight procedures have more freedom than might be apparent.  By use
appropriate agreements with procedures that call a lightweight procedure,
by appropriate agreements with procedures that a lightweight procedure calls,
and by 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.

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

3.2 Invocation Descriptor—Procedures With a Stack Frame

A procedure which allocates a call frame on the stack (sometimes referred to as a "heavy-weight procedure") can save and restore nonscratch registers, and may make standard calls to other procedures. A standard heavyweight procedure maintains the invocation context in stack storage as defined in section "The Call Stack".

```
                                                          Quadword Aligned
+---------------+---------------+---------------+---------------+
|                          CODE_ADDRESS                         | :0
|                           (32 bits)                           |
+---------------+---------------+---------------+---------------+
|        FP_SP_DIFFERENCE       | MUST BE ZERO  |   FLAGS  | FMT | :4
|           (16 bits)           |   (8 bits)    |    (5)   | (3) |
+---------------+---------------+---------------+---------------+
|                       CONDITION_HANDLER                       | :8
|                           (32 bits)                           |
+---------------+---------------+---------------+---------------+
|     VECTOR_REGISTER_MASK      |        REGISTER_OFFSET        | :12
|           (16 bits)           |           (16 bits)           |
+---------------+---------------+---------------+---------------+
|                                                               | :16
|                         REGISTER_MASK                         |
|                           (64 bits)                           | :20
|                                                               |
+---------------+---------------+---------------+---------------+
```

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

FMT = INVDESCR$C_STACK_FRAME for an invocation descriptor denoting a procedure with a stack frame.

FLAGS is a set of flag bits defined as follows.

- FLAGS<HAS_HANDLER> is 1 if this procedure has a condition handler. If FLAGS<HAS_HANDLER> is 1, then the CONDITION_HANDLER field must denote a condition handler.
- FLAGS<HANDLER_REINVOKABLE> is 1 if the condition handler is a reinvokable handler, that is, one which can be reinvoked should another condition occur while the handler is already active. If FLAGS<HANDLER_REINVOKABLE> is 0 then the condition handler is not reinvokable.

  FLAGS<HANDLER_REINVOKABLE> must be 0 unless FLAGS<HAS_HANDLER> is 1.
- FLAGS<VECTOR_CONTEXT_SAVE> is 1 if this procedure saves VL, VC, and VM in the register save area on entry to the procedure.
- FLAGS<3:4> must be 00.

FP_SP_DIFFERENCE is the distance in bytes between the stack frame base (this procedure's FP value) and the stack's top (SP value) at entry to the procedure (see section "Stack Frame Layout"). FP_SP_DIFFERENCE must be a multiple of 8 so as to maintain quadword-alignment of the stack.

CONDITION_HANDLER is a signed self-relative pointer to the invocation descriptor for a run time static condition handling procedure. This field is present in all invocation descriptors with FMT = INVDESCR$C_STACK_FRAME, but is *not* defined unless FLAGS specifies that there is a condition handler.

REGISTER_OFFSET is the difference in bytes between the stack frame base (this procedure's FP value) and the register save area (see section "Stack Frame Layout"). REGISTER_OFFSET must be a multiple of 8 such that REGISTER_OFFSET(FP) yields a quadword aligned address.

REGISTER_MASK is a bit vector (0..63) specifying the scalar registers that are saved in the register save area on entry to the procedure.

VECTOR_REGISTER_MASK is a bit vector (0..15) specifying the vector registers that are saved in the register save area on entry to the procedure.

```
\\
Dynamic condition handling is not defined by this calling standard;  it is
language-defined.  Compilers will set up language-specific static condition
handlers, and these static condition handlers will provide the dynamic
condition handling semantics of each language.

Users should not write static condition handlers, as they are part of the
software implementation architecture and are not necessarily portable to
other architectures.

Users should utilize the dynamic condition handling mechanisms provided by the
language in which they are coding, which are implemented and invoked via
compiler and RTL mechanisms, and are portable.

\\
```

4 The Linkage Section

Since a PRISM instruction cannot contain a full virtual address, PRISM is a base register architecture. All memory references are displacements relative to the contents of a base register (or implicitly, the program counter), and all external references must be indirect through address constants.

The fundamental table of address constants that a procedure can use to access static storage and external procedures and variables is called a *linkage section*. A register that is used to access the contents of the linkage section is a *linkage pointer*.

A procedure's linkage section includes the invocation descriptor for that procedure and the addresses of all external constants, variables, and procedures referenced by the procedure.

When a separately compiled procedure is called, the caller must provide a pointer (in R10) to the called procedure's invocation descriptor. R10, the linkage pointer, may be used by the called procedure as a base register to access address constants in its linkage section.

Linkages to external procedures are represented in the calling procedure's linkage section as a *linkage pair*. A linkage pair, which must be quadword aligned, consists of two longwords.

```
                                                  Quadword Aligned
+-----------------------------------------------------------------+
|                                                                 |
|                  invocation descriptor address                  | :0
|                                                                 |
+-----------------------------------------------------------------+
|                                                                 |
|                          code address                           | :4
|                                                                 |
+-----------------------------------------------------------------+
```

Invocation descriptor address is the absolute address of the invocation descriptor of the called procedure.

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

Although, because the invocation descriptor also contains the absolute code address, only the invocation descriptor address is strictly required in the linkage section, the absolute code address is placed in the linkage section to provide better performance for external procedure calls. Although the absolute code address in the invocation descriptor is used under certain circumstances (see section "Procedure Values"), this structure of linkage pairs in the linkage section is *required* by this standard—the second longword must always be present.

```
\\
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, which will
contain an invocation descriptor for each procedure in the module.  Offsets for
invocation 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 read-only PSECT. This read-only linkage PSECT could also contain all read-only constants defined by the module. However, when an image is activated and the fixups on the linkage section are performed, all the read-only constants in that PSECT become non-shared between processes. If there are more than a page or two of read-only constants, this may significantly increase physical memory usage; thus, caution must be exercised.

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

As an example of typical code to call an external procedure, Q, suppose that LP is a register currently containing the address of the current procedure's invocation descriptor (LP is not a fixed register). Q can be called by:

```
    LDQ    Q_OFFSET(LP), R10    ; Q's linkage pair into R10..R11
    JSR    R11, (R11)           ; Jump to Q.  Return address in R11
```

Because Q's invocation descriptor is in Q's linkage section, Q can use the value in R10 as a base address for accessing data in its linkage PSECT. Q accesses external procedures and data in other PSECTs through pointers in its linkage PSECT. Therefore, R10 serves as the root pointer for access to all static data.
\\

The following example illustrates the layout and use of linkage sections. (This example does *not* presume to represent how a compiler might actually layout a linkage section, or to represent actual PRISM assembler notation, and the code fragments presented do not presume to do anything useful.)

MODULE X

```
        .psect      $LINK         ; Linkage section for Module X
X1::                              ; Invocation descriptor for heavyweight procedure X1
        .address    X1_ENTRY      ; Address of entry code sequence for X1
        .byte         1           ; X1 has stack frame but no condition handler
        .byte         0           ; MBZ
        .word        32           ; FP_SP_DIFFERENCE = 32
        .long         0           ; No condition handler
        .word        16           ; REGISTER_OFFSET = 16
        .word         0           ; No saved V0..V15
        .quad         7           ; Save R61..R63

Y1_PROCVAL:                       ; Procedure value for procedure Y1
        .address    Y1            ; Value is Y1's invocation descriptor address

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    Y_DATA_1      ; Address of Y_DATA_1
Y_DATA_2_ADDR:
        .address    Y_DATA_2      ; Address of Y_DATA_2

        .psect      $CODE         ; Code for Module X

        ; Entry code sequence for procedure X1
```

```
X1_ENTRY:                          ; Address of X1's invocation descriptor is in R10
                                   ; on standard call, and return address is
                                   ; in R11
        LDA        -32(SP),SP      ; Allocate stack space for X1 stack frame
        STQ        R10,(SP)        ; Save invocation descriptor address and return address
        STL        FP,8(SP)        ; Save caller's frame pointer
        STQ        R62,16(SP)      ; Save R62..R63
        STL        R61,24(SP)      ; Save R61
        OR         R0,SP,FP        ; X1 is now the current procedure
        OR         R0,R10,R61      ; Move own invocation descriptor address to preserved
                                   ; register R61

        ; Calculate 2 raised to power of first argument and store in Y_DATA_1

        LDA        POWER_2_TABLE-X1(R61),R4
                                   ; R4  <- address of powers of 2 table
        SLL        #2,R14,R5
        ADD        R4,R5,R4        ; R4  <- address of value of 2**first argument
        LDL        (R4),R4         ; R4  <- 2**first argument
        LDL        Y_DATA_1_ADDR-X1(R61),R5
                                   ; R5  <- address Y_DATA_1
        STL        R4,(R5)         ; Y_DATA_1  <- R4

        ; Preserve arguments across call to Y1

        OR         R0,R14,R62      ; R62  <- first argument
        OR         R0,R15,R63      ; R63  <- second argument

        ; CALL procedure Y1

        LDQ        Y1_PROCVAL-X1(R61),R10
                                   ; R10  <- address of Y1's invocation descriptor
                                   ; R11  <- address Y1's entry code sequence
        JSR        R11,(R11)       ; R11  <- return address and call Y1

        ; Store first+second argument in external Y_DATA_2 and return sum
        ; as X1 value

        ADD        R62,R63,R8      ; R8  <- sum
        LDL        Y_DATA_2_ADDR-X1(R61),R4
                                   ; R4  <- address Y_DATA_2
        STL        R8,(R4)         ; Y_DATA_2  <- R8
        ; Return sum to caller.

        LDL        24(FP),R61      ; Restore R61
        LDQ        16(FP),R62      ; Restore R62..R63
        LDQ        4(FP),R4        ; Get return address and caller's frame pointer
        OR         R0,R5,FP        ; Restore caller's frame pointer.  Caller is once
                                   ; again the current procedure.
        LDA        32(R5),SP       ; Restore caller's SP
        JSR        R0,(R4)         ; Return to caller
-------------------------------------------------------------------------------
-------------------------------------------------------------------------------
```

MODULE Y

```
        .psect     $LINK           ; Linkage section for Module Y

Y1::                               ; Invocation descriptor for lightweight procedure Y1
        .address   Y1_ENTRY        ; Address of entry code sequence for Y1
        .byte        8             ; Y1 has register frame and a nonreinvokable handler
        .byte        7             ; Save caller's FP in R7
        .byte        1             ; Leave caller's SP undisturbed in R1
        .byte       11             ; Leave return address undisturbed in R11
        .long      Y1_HND-.        ; Y1_HND is condition handler
```

```
Y1_HND:                            ; Invocation descriptor for condition handler
        .address   Y1_HND_ENTRY ; Address of entry code sequence for Y1_HND.
                                   ; (Y1_HND_ENTRY is not shown in this example.)
        .byte        0             ; Y1_HND has register frame and no condition handler
        .byte        4             ; Save caller's FP in R4
        .byte        1             ; Leave caller's SP undisturbed in R1
        .byte       11             ; Leave return address undisturbed in R11
        .long        0             ; No condition handler

Y_STATIC_DATA:                     ; address of base of static Y data
        .address   Y_DATA_PSECT

        .psect     $DATA           ; static data psect

Y_DATA_PSECT:
Y_DATA_1::                         ; global static variable Y_DATA_1
        .long      1
Y_DATA_2::                         ; global static variable Y_DATA_2
        .long      1

        .psect     $CODE           ; Code for Module Y

        ; Entry code sequence for procedure Y1

Y1_ENTRY:                          ; Address of Y1's invocation descriptor is in R10
                                   ; on standard call, and return address is
                                   ; in R11
        OR         R0,FP,R7        ; Save caller's frame pointer
        DRAIN                      ; Force pending exceptions to be raised
        OR         #4,R10,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        Y_STATIC_DATA-Y1(R10),R4       ; R4 <- base of static data psect
        LDL        Y_DATA_1-Y_DATA_PSECT(R4),R5   ; R5 <- Y_DATA_1 value
        LDL        Y_DATA_2-Y_DATA_PSECT(R4),R6   ; R6 <- Y_DATA_2 value
        MUL        R5,R6,R5                       ; Calculate product
        DRAIN                                     ; Synchronize exceptions
        STL        R5,Y_DATA_1-Y_DATA_PSECT(R4)   ; Y_DATA_1 <- product

        ; return to caller

        OR         R0,R7,FP        ; Restore caller's frame pointer.  Caller is once
                                   ; again the current procedure.
        JSR        R0,(R11)        ; Return to caller
```

5 The PRISM Call Conventions

The PRISM call conventions determine the methods used to communicate certain data between the caller and the called procedure during invocation and return. Elements of these conventions may be optional, depending on the properties of the called procedure.

The register or stack location used for an element in standard calls is specified as part of the calling standard.

- **Invocation Descriptor Address**

  This is the address of the invocation descriptor of the called procedure. This address is required in standard calls and is contained in register R10.

- **Return Address**

  In a normal return from the called procedure, the return address is the address of the instruction to which control is transferred. In a standard call, this address is contained in register R11.

- **Argument Count**

  This is the number of longwords in the argument list. This count is required in a standard call and is contained in register R13.

- **Argument List**

  The argument list in a PRISM call is an ordered set of zero or more *argument items*, which together comprise a logically contiguous structure known as the *argument item sequence*. This logically contiguous sequence is in practice mapped to registers and memory in a fashion that may produce a physically discontiguous argument list. In a standard call, the first **N** items are passed in registers R14..R21. (See section "Argument Lists" for details of argument-to-register correspondence). The remaining items are collected in a memory argument list, which is a quadword aligned array of longwords. In a standard call, the address of this list (if present) is contained in register R12.

- **Function Result**

  If a standard-conforming procedure is a function, and the function result is to be returned in a register, then the result is returned in R8 or R8..R9. Otherwise, the function result is returned via the first argument item. (See section "Function Value Return").

- **Scratch Registers**

  These are registers that the called procedure is allowed to modify without saving and restoring their contents. In a standard call, these are R4..R31.

6 The Call Stack

6.1 Stack Usage, FP, and SP

The PRISM treatment of condition handling, unwinding (nonlocal GOTOs), trace back, and debugging depends on conventions governing the use of R1 (SP, the stack pointer register) and R2 (FP, the frame pointer register). SP is used by the hardware in raising exceptions and asynchronous software conditions.

SP must denote a quadword aligned address, that is, SP<2:0> must be 000.

The contents of the stack located at addresses lower than (SP) are continually and unpredictably modified. The contents of the stack located at addresses higher than FP_SP_DIFFERENCE(FP) belong to the calling procedure; they should not be read or written by the called procedure, except as specified in the argument list or by language-controlled up level references.

Additional rules for FP are more complex. Its value in a thread denotes the invocation context of the current procedure in that thread, which is the root of a chain of procedure invocation contexts, including a (possibly null) set of register values saved by each invocation. The first procedure in the invocation chain is the current procedure, the next is its predecessor (the procedure that invoked the current procedure), and so forth. The current and saved register values associated with each procedure invocation define the values of all registers at the moment of that invocation, excluding registers that are scratch registers for that procedure. The invocation chain information is thus sufficient to analyze a thread at any moment and to implement stack unwinding.

Given the current register values for a procedure invocation, that invocation can be interpreted by examining the value of FP.

If FP<2> is 0 then the current procedure has a stack frame, and FP<31:3> contains the address of the quadword aligned stack frame base. The standard stack frame layout specifies the location of key information in the frame, including the location of the address of the current procedure's invocation descriptor and the register values saved by that procedure. From this information the register values and context of the preceding invocation can be determined.

If FP<2> is 1 then the current procedure has a register frame, and FP<31:3> contains the address of the quadword aligned invocation descriptor. Information in the invocation descriptor specifies which current registers maintain the return address and the predecessor's FP and SP values; these are the only registers needed to determine the register values and context for the predecessor.

Compilers may use FP<1:0> for other flags. That is, FP<2:0> must always be masked to obtain the stack frame or invocation descriptor address.

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.

6.2 Stack Frame Layout

The following figure illustrates the layout of the PRISM stack frame for procedures that have stack frames. 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.

```
                                                 Quadword Aligned
+-------------------------------------------------------------+
|                                                             |  :(SP)
|                                                             |
|                                                             |
|                                                             |
|                  [stack-temporary area]                     |
|                                                             |
|                                                             |
|                                                             |
|                                                             |
+-------------------------------------------------------------+
|                                                             |
|                  INVOCATION_DESCRIPTOR                      |  :(FP)
|                                                             |
+-------------------------------------------------------------+
|                                                             |
|                     RETURN_ADDRESS                          |  :4(FP)
|                                                             |
+-------------------------------------------------------------+
|                                                             |
|                   SAVED_FRAME_POINTER                       |  :8(FP)
|                                                             |
+-------------------------------------------------------------+
|                                                             |  :12(FP)
|                                                             |
|                                                             |
|                  [other fixed locations]                    |
|                                                             |
|                                                             |
|                                                             |
+-------------------------------------------------------------+
|                                                             |  :REGISTER_OFFSET(FP)
|                                                             |
|                                                             |
|                                                             |
|                   [register save area]                      |
|                                                             |
|                                                             |
|                                                             |
|                                                             |
+-------------------------------------------------------------+
|                                                             |
|                                                             |
|                                                             |
|                  [other fixed locations]                    |
|                                                             |
|                                                             |
|                                                             |
+-------------------------------------------------------------+ :FP_SP_DIFFERENCE(FP)
```

The information needed to interpret the calling chain is at locations (FP), 4(FP), and 8(FP); these locations must be maintained as defined this calling standard. The stack frame must be allocated and initialized by the entry code sequence of a called procedure with a stack frame.

INVOCATION_DESCRIPTOR contains the address of the invocation descriptor of the current procedure.

RETURN_ADDRESS contains the address of the instruction to which control is to be transferred on a normal return.

SAVED_FRAME_POINTER contains the caller's FP.

Other fixed 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 condition handling context, fixed temporaries, etc.

```
\\
Since REGISTER_OFFSET(FP) must be quadword aligned, it can't be at 12(FP), so 12(FP)
must either be a fixed compiler temporary or unused.
\\
```

Register save area is a set of consecutive longwords in which nonscratch registers modified by the current procedure are saved. The register save area begins at REGISTER_OFFSET(FP), where REGISTER_OFFSET is specified in the procedure's invocation descriptor. REGISTER_OFFSET(FP) must yield a quadword aligned address. The set of registers saved is specified in the invocation descriptor, by the REGISTER_MASK and VECTOR_REGISTER_MASK fields and by FLAGS<VECTOR_CONTEXT_SAVE>.

The high-address end of the stack frame is defined by the value FP_SP_DIFFERENCE in the procedure's invocation descriptor. The high-address end is used to determine the value of SP for the predecessor procedure in the calling chain.

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.

6.3 The Register Save Area

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

1. All even-odd saved scalar register pairs are stored, in register-number order, in consecutive quadwords.
2. All even or odd saved scalar registers whose paired register is not being saved are stored, in register-number order, in consecutive longwords following the even-odd register pairs.
3. If an odd number of scalar registers are being saved, an additional longword must be allocated following the last saved scalar register.
4. All saved vector registers V0..V15 are stored, in register-number order, in consecutive quadwords beginning at the first quadword following the saved scalar registers.
5. If VM, VL, and VC are being saved,
    1. VM is stored in the first quadword following the last saved register.
    2. VL is stored in the longword following the saved VM.

3. VC is stored in the longword following the saved VL.

For example, if registers R40, R42, R43, R50, and R54 were to be saved, they would be packed in the register save area as follows.

```
                                                 Quadword Aligned
+-------------------------------------------------------------+
|                             R42                             | :REGISTER_OFFSET(FP)
+-------------------------------------------------------------+
|                             R43                             |
+-------------------------------------------------------------+
|                             R40                             |
+-------------------------------------------------------------+
|                             R50                             |
+-------------------------------------------------------------+
|                             R54                             |
+-------------------------------------------------------------+
|                           scratch                           |
+-------------------------------------------------------------+
```

If R51 had also been saved, the contents of the register save area would instead be

```
                                                 Quadword Aligned
+-------------------------------------------------------------+
|                             R42                             | :REGISTER_OFFSET(FP)
+-------------------------------------------------------------+
|                             R43                             |
+-------------------------------------------------------------+
|                             R50                             |
+-------------------------------------------------------------+
|                             R51                             |
+-------------------------------------------------------------+
|                             R40                             |
+-------------------------------------------------------------+
|                             R54                             |
+-------------------------------------------------------------+
```

7 Entry and Return Code Sequences

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

1. Allocate and initialize a stack frame (if a procedure with stack frame).
2. Initialize RA_SAVE, SP_SAVE, and FP_SAVE (if a procedure with a register frame)
3. Store all saved registers (if any).
4. Execute a DRAIN instruction (if the procedure has a condition handler)
5. Set FP in a manner consistent with the register conventions that define the calling chain.

The current procedure (as defined by the conventions) is the calling procedure until the called procedure's entry code sequence sets FP.

When a procedure returns, the return code sequence must

1. Restore saved registers (if any).
2. Execute a DRAIN instruction (if the procedure has a condition handler)
3. Restore FP to the value it had on entry to the procedure.
4. Reset SP (if modified by the procedure).

   SP will normally be reset to the value it had on entry to the procedure; however, in some cases the returning procedure must leave SP pointing to a lower stack address than it had on entry to the procedure (see section "Function Value Return").

In addition, if the called procedure executes vector loads and stores it must under certain circumstances execute a DRAINM to synchronize memory with the calling procedure or with procedures it calls (see section "Interprocedural Synchronization").

The examples below illustrate entry and return code sequences; there are many other possible code sequences depending on the called procedure's register use and frame layout, and the compiler's optimization methods.

All the examples assume that procedure Q is invoked by a standard call from procedure P.

7.1 Entry Code Sequence—Register Frame

For convenience, this example assumes that Q has no static condition handler, RA_SAVE specifies R11, and SP_SAVE specifies R1 (that is, the procedure allocates no local stack storage).

```
OR    R0, FP, FP_SAVE ; FP_SAVE is specified in Q's invocation descriptor.
OR    #4, R10, FP     ; FP = address (Q's invocation descriptor) OR 4
                      ; Q is now the current procedure.
```

7.2 Return Code Sequence—Register Frame

```
OR    R0, FP_SAVE, FP ; Restore P's FP value.
                      ; P is once again the current procedure.
JSR   R0, (R11)       ; Return to P's code.
```

7.3 Entry Code Sequence—Stack Frame

For convenience, this example assumes that the regions for language-specific fixed locations are empty, so that REGISTER_OFFSET = 16 and when the entry code sequence is complete FP = SP.

```
LDA    -Q_FP_SP_DIFFERENCE(SP), SP
                             ; Allocate space for new stack frame.
STQ    R10,(SP)              ; Save address of invocation descriptor and
                             ; return address
STL    FP, 8(SP)             ; Save caller's FP.
STL    Rx, 16(SP)            ; Save first register.
...
DRAIN                        ; Force any pending hardware exceptions
                             ; to be raised.  Required if Q has a
                             ; condition handler so that a pending exception
                             ; caused by the caller will not be raised in the
                             ; context of Q.
OR      R0, SP, FP           ; Q is now the current procedure.
```

Note that if this code sequence is interrupted by an asynchronous software condition, 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 the condition handler result in a nonlocal GOTO to a location in the procedure, P, that called Q, then it will not be possible to restore SP to the correct value in P.

Therefore, any procedure that contains a label which can be the target of a non-local GOTO must reset SP at that label.

7.4 Return Code Sequence—Stack Frame

This is the return code sequence for the preceding example. The example below assumes the return address is still in R11.

```
...                          ; Restore saved registers.
LDL    16(FP), Rx            ; Restore first register saved.
DRAIN                        ; Force any pending hardware exceptions
                             ; to be raised.  Required if Q has a
                             ; condition handler so that a pending exception will
                             ; be raised in the context of Q.
OR     R0,FP,SP              ; Remove temporary stack storage.
LDL    8(FP), FP             ; Restore FP.  P is once again the current procedure.
LDA    Q_FP_SP_DIFFERENCE(SP), SP
                             ; Restore SP.
JSR    R0, (R11)             ; Return to P.
```

Interruption of this code sequence by an asynchronous software condition can result in P being the current procedure but with SP not yet restored to its value in P. The discussion of that situation in entry code sequences applies here as well.

8 Procedure Values

PRISM procedure values utilize the properties of invocation descriptors.

A PRISM procedure value is not a procedure's entry address, and it is not necessarily the address of an actual invocation descriptor. Rather, a procedure value is a pointer that can be *treated* as the address of an invocation descriptor. That is, a procedure value points to a data structure whose first longword contains the address to which the calling procedure must jump. The procedure value itself must be passed as part of the calling sequence.

Suppose register R32 contains a procedure value. An example of the code to call the procedure is:

```
OR     R0, R32, R10 ; Procedure value to R10
LDL    (R32), R11   ; Entry address to scratch register
JSR    R11, (R11)   ; Jump to entry address;  return address goes in R11
                    ; The use of R10 and R11 to is specified by the calling standard.
                    ; R11 is also used here as a temporary.
```

If a procedure Q is not nested within another procedure, then a procedure value for Q is simply the address of Q's invocation descriptor, and the code sequence above calls Q.

If Q is a subprocedure of a procedure P, then a procedure value for Q must be a bound procedure value. The bound procedure value is a data structure of three longwords that provides a parent frame pointer, the address of an invocation descriptor, and the address of a transfer code sequence.

```
                                                   Quadword Aligned
+----------------------------------------------------------+
|                                                          |
|             address of transfer code sequence            |:0
|                                                          |
+----------------------------------------------------------+
|                                                          |
|             address of invocation descriptor             |:4
|                                                          |
+----------------------------------------------------------+
|                                                          |
|                   parent frame pointer                   |:8
|                                                          |
+----------------------------------------------------------+
```

When the transfer code sequence addressed by the first longword is called (by a call sequence such as the one above), the procedure value will be in R10, and the transfer code must finish setting up the elements of the standard call.

An example of a such a transfer code sequence, for a target procedure that expects the parent frame pointer to be passed in R30, is:

```
LDL   8(R10), R30   ; Parent frame pointer to R30
LDL   4(R10), R10   ; Invocation descriptor address to R10
LDL    (R10), R4    ; Entry address to scratch register
JSR   R0, (R4)      ; Jump to entry address.
                    ; Return address was already in R11, and this code
                    ; sequence preserves the return address in R11.
```

Here, when control is transferred to Q's entry address, R10 contains the address of Q's invocation descriptor, R11 contains the return address, and R30 contains the parent frame pointer (which is the parent call frame pointer, FP, for the invocation of P to which Q is bound in this procedure value).

The parent frame pointer 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 data structure will normally be allocated in the parent frame.

Procedure values as defined here are typically not used to manage a call within a module; in that situation the call may be done by a PC-relative JSR instruction.

```
\\
Note that bound procedure values must be generated at run-time.  The parent
frame pointer is not known until the parent has been invoked.  In addition,
if a bound procedure value were to be statically initialized, then the
PRISM Common Software Architecture would require it to be four longwords, with
the absolute address of the target procedure's entry code sequence in the third
longword -- that is, a bound procedure value would have to have a linkage pair
embedded within it.
\\
```

9 Argument Passing Mechanisms

The term *argument item* means an item passed by PRISM call conventions, representing an argument to be associated with a corresponding parameter in the called procedure.

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

- Immediate value
- Reference
- Descriptor

The standard permits any combination of these mechanisms in an argument list. Argument items are not self-defining: interpretation of each argument item depends on agreement between the calling and called procedures.

9.1 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.

An argument may be passed by immediate value only if

- it is a scalar data type with known size $\leq$ 32 bits,

  or
- it is a record with known size $\leq$ 32 bits.

  or
- it is a set with known size $\leq$ 32 bits.

No form of string or array may be passed by immediate value.

A standard immediate argument item must be a longword. (This standard also defines the rules for passing immediate arguments > 32 bits, but such arguments may not be used in standard calls).

The unused high-order bits of all data types (including records) must be zero-extended or sign-extended, to the next longword boundary, as appropriate.

9.2 Reference

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

A reference argument item must be a longword.

9.3 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, or the value contained in it, is to be associated with the parameter.

A descriptor argument item must be a longword, and the descriptor to which it points must be quadword aligned.

```
\\
The rules that determine how high level language arguments are mapped to these
argument passing mechanisms are language defined.  In general,
languages should use the same rules as those used on VAX, in order to avoid
pertubations in the user-visible programming environment from VAX to PRISM, and
to maintain interlanguage compatibility.

In practice, this means by reference except when descriptor is required.

System implementation languages (PILLAR and C) will, however, use immediate
whenever possible.
\\
```

10 Argument Lists

10.1 Argument List Structure

The *argument list* in a PRISM 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* in a standard procedure call must be a longword. A longword argument item may be used to pass immediate arguments ≤ 32 bits, arguments by reference, and arguments by descriptor.

Although the longword argument items form a logically contiguous sequence, they are in practice mapped to registers and memory in a fashion that may produce a physically discontiguous argument list. Registers R14..R21 are used to pass the first eight longwords of the argument item sequence. Additional argument items must be passed in a quadword aligned memory argument list, the address of which must be passed in R12.

Argument items are assigned to registers R14 to R21 and to longwords in the memory argument list according to the following rules:

1. Argument items are assigned in order to increasing argument registers, beginning with R14.
2. If there is one or more unassigned argument items after argument register R21 has been allocated, then all remaining argument items are assigned to the memory argument list in order from the lowest-addressed longword to the highest-addressed longword.
   - The address of the memory argument list must be passed in R12.
   - The memory argument list must be quadword aligned.
3. Argument items must not be directly assigned to R12.

That is, the order of the arguments in registers and memory is R14 < R21 < (R12) < (R12)+N.

The memory portion of the argument list must be treated as read-only data by the called procedure, and may be allocated in read-only memory at the option of the calling procedure (except by agreement between the calling and called procedure, such as for output parameters).

10.2 Large Immediate Arguments

Certain languages will pass immediate arguments > 32 bits as *large immediate* arguments.

Such arguments are *not* standard, cannot be used for interlanguage procedure calls, and must *not* be used in public interfaces callable by multiple languages. However, this standard defines how large immediate arguments must be passed by languages that support them.

Large immediate arguments are treated as a sequence of longwords, and are assigned to argument registers and memory as though they were a sequence of different argument items. This means that large immediate argument is not necessarily quadword aligned, and may be split between R21 and (R12).

Large immediate arguments that are not a multiple of 32 bits in length must be zero-extended or sign-extended to the next multiple of 32 bits.

The sequence of longwords comprising a long immediate argument is assigned to registers and memory according to the algorithm in the preceeding section, where the lowest addressed longword of the immediate argument is the first argument item contributed by the immediate argument, the next-lowest addressed longword of the immediate argument is the next argument item contributed by the immediate argument, and so on.

Large immediate arguments are treated as a number of separate longword arguments when computing the argument count to be stored in R13.

10.3 Argument Lists and High Level Languages

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

1. Arguments are mapped from left to right to increasing offsets in the argument item sequence. R14 is allocated to the first argument, and the last longword of the memory argument list (if any) is allocated to the last argument.
2. Each source language argument corresponds to a single PRISM argument item, except for certain parameterized types.

   For parameterized types, if the source language argument is not being passed by a single standard descriptor, then the address of the argument value is passed in a single argument item (that is, the argument value is passed by reference) and the type parameter values are passed in consecutive argument items immediately following the argument value item in the argument item sequence.
3. Each argument item is a longword.

   This may require zero-extension or sign-extension.
4. A null or omitted argument, for example CALL SUB(A,,B), is represented by a longword 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 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.

10.4 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.

11 Procedure Argument Descriptors

11.1 Goals

- Support high level languages

  Support passing parameters between procedures that conform to the PRISM calling standard. Thus, they are designed to support high level language (HLL) procedures calling procedures which are:

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

- Provide architectural extensibility

  Two variants of the PRISM architecture are anticipated, 32-bit and 64-bit. Because of this, the logical and physical attributes of descriptors must be separated.

  - The information found in a PRISM descriptor will be the same across all variants of the PRISM architecture, and the user's view of PRISM descriptors should not be sensitive to the variants of the architecture. This is termed the *logical view* of the architecture.
  - The *physical view* of a descriptor includes the order and the size of each field. The size and order of the fields will not necessarily be the same across all variants of the PRISM architecture.

  Definitions of descriptors at both the logical and physical level are provided. Only routines using the logical view will work across all variants of the architecture. Routines using the physical view will *not* work across all variants of the architecture.

  Software that needs to view descriptor fields should use the logical view wherever possible so that it remains invariant across the PRISM architecture. High level language users should utilize descriptors via mechanisms such as %DESCR, attributes on variables, and macros or lightweight procedures for accessing and manipulating descriptor attributes.

- VAX compatibility

  VAX programs that use descriptors via mechanisms such as %DESCR will continue to work on PRISM (*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. PRISM descriptors favor improving the speed of access to data.

- Avoid redundant information

  Several VAX descriptors contain fields that can be derived from other fields in the descriptor. PRISM descriptors are designed to avoid that redundant information.

- Avoid redundant forms.

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

- Support 8-bit and multibyte character sets

  PRISM descriptors support 8-bit character sets and multibyte character sets.

- Avoid unsupported combinations of fields.

  PRISM Descriptors are designed to avoid unsupported combinations of fields. VAX 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 was applied to PRISM.

- Provide uniform descriptors for procedure arguments, function return values, and condition arguments.

- Provide uniform array descriptors

  On VAX, 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 on PRISM 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 uniformity descriptors for sequences of bits and characters.

  On VAX, 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 on PRISM by having one form of descriptor for linear, one-dimensional data.

- Avoid addressing calculations in called procedures.

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

11.2 Non-Goals

- Descriptors are not designed to describe objects to the degree needed by tools such as the PRISM Debugger or PRISM CDD.

  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 and PRISM descriptors are not identical.

  PRISM descriptors support high level language use of descriptors. Code that explicitly declares and references fields in VAX descriptors is unlikely to work on PRISM.

11.3 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 a PRISM 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).

Descriptors must be quadword aligned.

Procedure argument descriptors are divided into two broad categories, each of which is discussed in a following subsection:

- descriptors for byte addressable data
- descriptors for bit addressable data

11.4 Descriptors for Byte Addressable Data

The broad class of descriptors for byte addressable data contains the following specific descriptor classes:

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

PRISM 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 byte addressable data descriptor classes support all the types of text strings understood in the PRISM 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.

11.4.1 Logical Attributes of Byte Addressable Data Descriptors

CLASS is the descriptor class attribute:

DESCR$C_FIXED_TEXT
DESCR$C_DYNAMIC_TEXT
DESCR$C_VARYING_TEXT
DESCR$C_BYTE_ARRAY

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 (low_bound(1), . . . low_bound(n)).

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

When CLASS = DESCR$C_FIXED_TEXT or CLASS = DESCR$C_DYNAMIC_TEXT, EXTENT is an integer $0..2^{31} - 1$ specifying the length in bytes of the string.

When CLASS = DESCR$C_VARYING_TEXT, EXTENT is an integer $0..2^{16} - 1$ specifying the maximum length in bytes of the varying string.

When CLASS = DESCR$C_BYTE_ARRAY, and DIMENSIONS = 1, and STRIDE = ELEMENT_SIZE, then EXTENT is an integer $0..2^{31} - 1$ specifying the length in bytes of the storage occupied by the array.

For all other cases, EXTENT is *undefined*.

DIMENSION is an unsigned integer 1..255 specifying the number of dimensions in an array. If CLASS is not DESCR$C_BYTE_ARRAY, then DIMENSION must be zero.

ELEMENT_SIZE specifies the number of bytes of storage occupied by each element of a character array, and is present if and *only* if CLASS = DESCR$C_BYTE_ARRAY.

For arrays of fixed length text strings, ELEMENT_SIZE is an integer $0..2^{31} - 1$ specifying the length in bytes of each element of the array.

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

For all other data types, ELEMENT_SIZE is *undefined*.

VIRTUAL_ORIGIN is the address of element (0, . . . 0), and is present if and *only* if CLASS = DESCR$C_BYTE_ARRAY. This address is not necessarily within the storage of the array.

If, and *only* if, CLASS = DESCR$C_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.

11.4.2 Data Type Conventions for Byte Addressable Data Descriptors

Byte addressable data 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 the character size.

11.4.3 Run Time Conventions for Byte Addressable Data Descriptors

The conventions controlling the use of byte addressable data descriptors are determined by the class of the descriptor.

- DESCR$C_FIXED_TEXT

  A fixed text string is sequence of $0..^{31} - 1$ bytes with a fixed extent.

  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.

- DESCR$C_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.

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

  POINTER specifies the address of the unsigned 16-bit integer count which preceeds the fixed length area. This integer, which must be ≤ 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.

- DESCR$C_DYNAMIC_TEXT

  A dynamic text string is a sequence of $0..2^{31} - 1$ bytes.

  An uninitialized DYNAMIC_TEXT descriptor, and a descriptor for a null dynamic string, has an EXTENT of zero.

  The EXTENT and 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 EXTENT and POINTER fields of the descriptor. With one exception, these system supplied functions are the *only* procedures that may modify to these fields.

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

- DESCR$C_BYTE_ARRAY

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

  All elements of an array must reside in distinct storage. That is, a PRISM 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

11.4.4 Physical Format of PRISM-32 Byte Addressable Data Descriptors

```
                                                       Quadword Aligned
+---------------+---------------+---------------+---------------+
|                            EXTENT                             | :0
+---------------+---------------+---------------+---------------+
|                            POINTER                            | :4
+---------------+---------------+---------------+---------------+
|             zero              |  DIMENSION#   |     CLASS     | :8
+---------------+---------------+---------------+---------------+
|                        VIRTUAL_ORIGIN*                        | :12
+---------------+---------------+---------------+---------------+
|                         ELEMENT_SIZE*                         | :16
+---------------+---------------+---------------+---------------+
|                         LOW_BOUND(1)*                         | :20
+---------------+---------------+---------------+---------------+
|                        HIGH_BOUND(1)*                         | :24
+---------------+---------------+---------------+---------------+
|                          STRIDE(1)*                           | :28
+---------------+---------------+---------------+---------------+
                             .......
+---------------+---------------+---------------+---------------+
|                         LOW_BOUND(n)*                         | :x
+---------------+---------------+---------------+---------------+
|                        HIGH_BOUND(n)*                         | :x+4
+---------------+---------------+---------------+---------------+
|                          STRIDE(n)*                           | :x+8
+---------------+---------------+---------------+---------------+

 # : must be zero unless CLASS = DESCR$C_BYTE_ARRAY

 * : present if and only if CLASS = DESCR$C_BYTE_ARRAY

 x = 12*(n - 1) + 20
```

11.5 Descriptors for Bit Addressable Data

The broad class of descriptors for bit addressable data contains the following specific descriptor classes:

- Bit string
- Bit addressable array

PRISM bit addressable data 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 data 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.

PRISM 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 bit addressable data descriptor classes support all the types of bit strings understood in the PRISM 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.

11.5.1 Logical Attributes of Bit Addressable Data Descriptors

CLASS is the descriptor class attribute:

DESCR$C_FIXED_BIT
DESCR$C_BIT_ARRAY

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

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

When CLASS = DESCR$C_FIXED_BIT, EXTENT is an integer $0..2^{31} - 1$ specifying the length in bits of the string.

When CLASS = DESCR$C_BIT_ARRAY, and DIMENSIONS = 1, and STRIDE = ELEMENT_SIZE, then EXTENT is an integer $0..2^{31} - 1$ specifying the length in bits of the storage occupied by the array.

For all other cases, EXTENT is *undefined*.

DIMENSION is an unsigned integer 1..255 specifying the number of dimensions in an array. If CLASS is not DESCR$C_BIT_ARRAY, then DIMENSION must be zero.

POSITION is a signed integer 0..31 specifying the relative bit position with respect to POINTER of the first bit of the string or, for an array, the relative bit position with respect to POINTER of element (low_bound(1), . . . low_bound(n)).

ELEMENT_SIZE specifies the number of bits of storage occupied by each element of a bit array, and is present if and *only* if CLASS = DESCR$C_BIT_ARRAY.

If, and *only* if, CLASS = DESCR$C_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.

11.5.2 Run Time Conventions for Bit Addressable Data Descriptors

The conventions controlling the use of bit addressable data descriptors are determined by the class of the descriptor.

- DESCR$C_FIXED_BIT

  A fixed bit string is sequence of $0..^{31} - 1$ bits with a fixed extent. 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 32 bits to represent.

  The POINTER-relative bit offset of the last bit in a bit string can always be represented as a signed 32-bit integer $\leq 2^{31} - 1$. This means that the extent passed in a a bit string descriptor array must be $\leq 2^{31} - 32$.

- DESCR$C_BIT_ARRAY

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

  All elements of an array must reside in distinct storage. That is, a PRISM 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 32 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 POINTER-relative bit offset of the last bit in the last element in a bit array must be $\leq 2^{31} - 1$. This means that the difference between absolute bit addresses of the first and last bits in an array must be $\leq 2^{31} - 32$.

11.5.3 Physical Format of PRISM-32 Bit Addressable Data Descriptors

```
                                                    Quadword Aligned
+--------------+--------------+--------------+--------------+
|                          EXTENT                           | :0
+--------------+--------------+--------------+--------------+
|                          POINTER                          | :4
+--------------+--------------+--------------+--------------+
|     zero     |-  POSITION   |  DIMENSION#  |    CLASS     | :8
+--------------+--------------+--------------+--------------+
|                       ELEMENT_SIZE*                       | :12
+--------------+--------------+--------------+--------------+
|                       LOW_BOUND(1)*                       | :16
+--------------+--------------+--------------+--------------+
|                       HIGH_BOUND(1)*                      | :20
+--------------+--------------+--------------+--------------+
|                       STRIDE(1)*                          | :24
+--------------+--------------+--------------+--------------+
                          .......
+--------------+--------------+--------------+--------------+
|                       LOW_BOUND(n)*                       | :x
+--------------+--------------+--------------+--------------+
|                       HIGH_BOUND(n)*                      | :x+4
+--------------+--------------+--------------+--------------+
|                       STRIDE(n)*                          | :x+8
+--------------+--------------+--------------+--------------+

  # : must be zero unless CLASS = DESCR$C_BIT_ARRAY

  * : present if and only if CLASS = DESCR$C_BIT_ARRAY

  x = 12*(n - 1) + 16
```

11.6 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 which accepts a fixed text string or a contiguous one-dimensional array of characters, either of the following descriptors may be passed:

CLASS = DESCR$C_FIXED_TEXT
EXTENT = *n*

or

CLASS = DESCR$C_BYTE_ARRAY
EXTENT = *n*
DIMENSION = 1
ELEMENT_SIZE = *character size*
HIGH_BOUND = EXTENT/ELEMENT_SIZE + LOW_BOUND − 1
STRIDE = ELEMENT_SIZE

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

CLASS = DESCR$C_FIXED_BIT
EXTENT = *n*

or

CLASS = DESCR$C_BIT_ARRAY
EXTENT = *n*
DIMENSION = 1
ELEMENT_SIZE = 1
HIGH_BOUND = EXTENT + LOW_BOUND − 1
STRIDE = 1

Any called procedure which accepts a FIXED_TEXT descriptor is allowed to interpret a BYTE_ARRAY descriptor as a FIXED_TEXT descriptor. Likewise, any called procedure which 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.

12 Function Value Return

A standard function must return its function value by one of three 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. Data types not supported by these mechanisms, such as noncontiguous arrays and variable-sized records, cannot be function values of a standard procedure.

12.1 Function Value Return By Immediate Value

A function value is returned by *immediate value* in register R8 or in R8..R9 if, and *only* if,

- it is a scalar data type with known size $\leq$ 64 bits,

  or
- it is a record with known size $\leq$ 64 bits.

  or
- it is a set with known size $\leq$ 64 bits.

No form of string or array may be returned by immediate value.

A function value is returned in register R8 if its data type is represented in $\leq$ 32 bits. Data types shorter than 32 bits must be zero-extended or sign-extended, as appropriate, to a full longword.

A function value is returned in R8..R9 if its data type is represented in $>$ 32 bits and $\leq$ 64 bits. Data types shorter than 64 bits must be zero-extended or sign-extended, as appropriate, to a full quadword. Two separate 32-bit entities cannot be returned in R8..R9.

12.2 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, 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.

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

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

12.3 Function Value Return By Descriptor

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

- the actual length of a function value is not known,

  and

- the function value is a string or a contiguous one-dimensional array.

No form of record, or any noncontiguous array or array with more than one dimension may be returned by descriptor.

The actual-argument list and the formal-argument list are shifted to the right by one argument item. The calling procedure must pass the address of a descriptor as the new, first argument item.

There are two distinct mechanisms for returning a function value by descriptor. The mechanism to be used is chosen by the calling procedure, and is specified by the class of the descriptor passed as the first argument item.

Any standard-conforming function which returns its value by descriptor must be prepared to handle both mechanisms, and must return its value according to the mechanism chosen by the caller.

12.3.1 Logical Attributes and Use of Fixed Buffer Return Descriptors

The fixed buffer mechanism supports returning the function value in storage provided by the caller for that purpose.

The caller must pass as the first argument a function return descriptor which specifies an existing region of contiguous storage. The fields of the descriptor must be initialized by the caller as follows.

CLASS = DESCR$C_FIXED_RETURN
POINTER = the address of the first byte of storage
EXTENT = a signed integer $0..2^{31}-1$ specifying the length in bytes of the storage

The called function must return its value as follows.

1. Write the return value into the storage specified by the descriptor, truncating the return value if its length exceeds the value of EXTENT.
2. Set register R8 to the length of the function return value *before any truncation*.

The called function must not modify the descriptor, and the descriptor may be allocated in read-only storage by the caller.

12.3.2 Physical Format of PRISM-32 Fixed Buffer Return Descriptors

```
                                                Quadword Aligned
+---------------+---------------+---------------+---------------+
|                             EXTENT                            | :0
+---------------+---------------+---------------+---------------+
|                             POINTER                           | :4
+---------------+---------------+---------------+---------------+
|                      zero                     |     CLASS     | :8
+---------------+---------------+---------------+---------------+
```

12.3.3 Logical Attributes and Use of Stack Return Descriptors

The top of stack mechanism supports returning the function value in stack storage allocated by the called function.

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

CLASS = DESCR$C_STACK_RETURN
POINTER = *undefined*
EXTENT = *undefined*

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.
   CLASS = DECSR$C_STACK_RETURN
   POINTER = the address of the first byte of the return value
   EXTENT = a signed integer $0..2^{31} - 1$ specifying the length in bytes of the return value
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 (SP). The return value must be entirely contained by stack storage at or above (SP).

No information is returned in registers R8..R9.

When control returns to the calling procedure, 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.

12.3.4 Physical Format of PRISM-32 Stack Return Descriptors

```
                                                Quadword Aligned
+---------------+---------------+---------------+---------------+
|                             EXTENT                            | :0
+---------------+---------------+---------------+---------------+
|                             POINTER                           | :4
+---------------+---------------+---------------+---------------+
|                      zero                     |     CLASS     | :8
+---------------+---------------+---------------+---------------+
```

12.3.5 Run Time Conventions for Function Value Return Descriptors

Functions which return a value by descriptor *must* use the following algorithm to determine which of the two mechanisms to use.

1. If CLASS = DESCR$C_FIXED_RETURN, then use the fixed buffer mechanism.
2. Otherwise, use the top of stack mechanism.

It is important that functions do not specifically test for CLASS = DESCR$C_STACK_RETURN. This is because future extensions to this standard may specify new function return mechanisms which define new CLASS codes, but which are upward compatible with the top of stack mechanism specified here (see Appendix C).

Any function which tests for CLASS = DESCR$C_STACK_RETURN will not be upward compatible when this standard is extended.

For the same reason, functions which return a value by the top of stack mechanism must *always* set CLASS = DESCR$C_STACK_RETURN.

Any function which returns a value by the top of stack mechanism and does not set CLASS = DESCR$C_STACK_RETURN will not be upward compatible when this standard is extended.

13 Data Types, Alignment, and Conventions for Interlanguage Use

13.1 Data Types

\\\\
This subsection will enumerate all of the data types for interlanguage use on PRISM and rigorously define them.

This section is still under development, pending completion of the Corporate Remote Procedure Calls Architecture, with which the data type architecture of this standard will interact.
\\\\

13.2 Data Alignment

A *natural address* is a virtual address which is multiple of at least the size of the data type, in bytes, which is being referenced. *Natural alignment* refers to placement of data such the first byte of the data is at a natural address for that data.

There are five natural alignments defined by this standard:

- Bit—any bit boundary
- Byte—any byte address
- Word—any byte address which is a multiple of 2
- Longword—any byte address which is a multiple of 4
- Quadword—any byte address which is a multiple of 8

On some implementations of the PRISM architecture, memory references to data which is not naturally aligned will result in alignment faults, which can degrade the performance of all procedures that reference the unnaturally aligned data.

For this reason, all data on PRISM systems must be aligned on natural boundaries. For example, 8-bit character strings must start on byte boundaries; 16-bit integers must start at addresses which are a multiple of at least 2 (word alignment); F-floating real values must start at addresses which are a multiple of at least 4 (longword alignment); G-floating real values must start at addresses which are a multiple of at least 8 (quadword alignment); and so forth.

Data types larger than 64 bits may use quadword alignment, which is the largest alignment defined by this standard. Alignments larger than quadword are language specific or application defined.

For aggregates such as strings, arrays, and records, the data type for purposes of alignment is *not* the aggregate itself, but the elements of which the aggregate is composed, and 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 which 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.

```
\\
This is the basic rule.  When we enumerate and define the PRISM data types, we
will also enumerate the alignment requirements of the individual data types.
\\
```

13.3 Record Layout Conventions

The PRISM record layout rules are designed to provide good run time performance on all implementations of the PRISM architecure, and to provide the required level of compatibility with VAX.

The PRISM calling standard therefore defines two record layout conventions:

- record layout conventions optimized for the PRISM architecture, referred to as *PRISM preferred* record layouts;
- record layout conventions which are compatible with those traditionally used by VAX languages, referred to as *VAX compatible* record layouts.

PRISM high level languages 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 not standard.

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

13.3.1 PRISM Preferred Record Layout

The PRISM 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 that they may be embedded in;
- 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 PRISM and VAX languages.

The PRISM 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. 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 preceeding 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 natural boundary for the data type; any unused bits following the last-used bit in the last-used byte of each component must be filled out to the next natural boundary such that any following data starts on a natural boundary.

   Strings and arrays must be aligned according to the natural alignment requirements of the data type of which the string or array is composed.
4. Records and subrecords must be aligned according to the largest natural alignment requirements of the contained elements and subrecords.

13.3.2 VAX Compatible Record Layout

The VAX 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 preceeding 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 immeidately 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.

14 Status Codes and Condition Values

```
\\
   TBS
\\
```

15 Condition Handling

A *condition* results from an event that occurs during thread execution, which the normal flow of control in the current procedure is not prepared to respond to. Such an event may be caused by hardware exceptions, such as arithmetic overflows, memory access violations, etc; or by actions performed in software, such as subscript range checking and assertion checking.

A *condition handler* is a procedure which is designed to handle conditions when they occur during the execution of the thread. When a condition occurs, the normal flow of control in the current procedure will be interrupted, the context saved, and a search initiated for a condition handler established by that thread.

If a handler is located, it will be called as a procedure with arguments that describe the nature of the condition and the environment within which the condition was raised. The handler may respond to the condition in several ways, including various combinations of the following.

- Perform some action that affects the context of the thread.
- Modify or augment the description of the condition.
- Raise a *nested condition*: cause another condition to occur in the context of the condition handler or in a procedure called directly or indirectly by the handler.

When a condition handler has completed all processing, it *completes* in one of the following ways.

- *Reraise* the condition: cause the search to be continued for another handler for this condition.
- *Continue* execution: resume execution of the thread at the point that it was interrupted.
- *Unwind*: resume execution of the thread at a point different than the point at which it was interrupted.
- *Exit Unwind*: cause thread execution to terminate.

This standard defines the interfaces, data structures, and algorithms that provide for reliable, controlled user mode handling of conditions on PRISM systems: the types of condition handlers, how they are established and invoked, how they may complete, the order in which they are invoked, and other properties of condition handling.

15.1 Goals

- Provide programmers and programs with reliable control over response to conditions and reporting of conditions, and over the flow of control when conditions occur.
- Provide user mode condition handling that is compatible and common across both the PRISM ULTRIX and PRISM MICA operating systems.
- Support the condition handling requirements of PRISM languages and layered products
- Provide for correct and predictable condition handling in a multilanguage environment, and the construction of modular, maintainable applications.

- Add no space or time overhead to procedure calls and returns that do not establish condition handlers and, and minimize time overhead for establishing handlers at the cost of increased time overhead when conditions occur.
- Provide condition handling for lightweight procedures that do not have a stack frame.
- Provide condition handling on a per-thread basis. Threads may not affect each other's condition handling.
- Provide subsystems and applications with the ability to override system messages to provide a more suitable application oriented interface.
- Maintain a degree of high level language source level compatibility with the VAX/VMS condition handling. Where possible, the majority of existing VMS applications should not need any logic or any drastic source changes to use PRISM condition handling.

  (There are some bounds on this goal due to the PRISM architecture).
- Define a superset of VAX/VMS condition handling capabilities, removing some past restrictions and adding useful capabilities for languages and run time libraries.

15.2 Conditions

A condition results from an event that occurs during thread execution, which the normal flow of control in the executing procedure is not prepared to respond to.

There are two classes of conditions in PRISM systems: *hardware conditions* and *software conditions*.

Hardware and software conditions are raised and handled in user mode with the same interfaces, data structures, and algorithms as hardware conditions. That is, there is unified condition handling for hardware and software conditions.

15.2.1 Hardware Conditions

Hardware conditions occur when a thread attempts some action defined as incorrect, impossible, or not yet possible by the hardware. Such action results in a *hardware exception* interrupting execution of the thread, which in turn causes a condition to be raised in context of the current procedure.

Hardware exceptions are always raised in the same thread that issued the instruction responsible for the exception.

The hardware exceptions which can result in a user mode condition on PRISM systems include

- Arithmetic traps
- Data alignment exceptions
- Faults occurring as a consequence of an instruction
- Memory management faults
- Serious system failures
- Stack alignment aborts
- Vector exceptions

The state of the machine when a hardware exception occurs, the exception-related information which is delivered to a user mode thread, and circumstances under which execution can be continued are all specific to individual hardware exceptions. Hardware exceptions are fully defined by the PRISM hardware architecture, which should be consulted for additional information.

15.2.2 Software Conditions

Software conditions result from an explicit action by a thread.

Software conditions may be raised at any point during thread execution. Applications and language run time libraries may utilize software conditions to notify a thread that some action defined as incorrect, impossible, or not yet possible was attempted by the thread. For example, subscript range checking failures and assertion checking failures may be raised as software conditions.

Software conditions may occur synchronously and asynchronously to thread execution. An asynchronous software condition is one which is raised in a thread by a different thread executing in the same or in a different process. (Asynchronous software conditions are not defined in this section; see section "Asynchronous Software Conditions".)

15.2.3 FLBC Instruction Faults

The Fault On Bit Clear (FLBC) instruction is available as a general mechanism for software to declare a condition without the usual overhead of setting up a procedure call. The displacement field of the FLBC instruction may be used to specify an argument for the condition.

Thus, a hardware fault exception may be used to raise what are essentially software conditions.

There are special software conventions that apply to the FLBC displacement argument, as defined in the following section.

15.3 Raising Software Conditions

15.3.1 Raising Synchronous Software Conditions

A thread may raise a synchronous software condition in its own context by calling a system supplied function.

Each PRISM system provides concrete language bindings to this function, which is abstractly defined in this standard as RAISE_CONDITION.

RAISE_CONDITION accepts one argument:

- CONDITION_RECORD: a primary condition record.

RAISE_CONDITION returns a status value, and only returns to the caller if a handler returns STATUS$_CONTINUE, or if an error is detected in the condition record argument.

15.3.2 FLBC Instruction Faults

```
\\
This will be defined in the next revision of this standard.
\\
```

15.4 Condition Handlers

There are four types of condition handlers:

- *Primary vectored* handlers
- *Invocation-based* handlers
- *Last chance vectored* handlers
- The system *catchall* handler

With the exception of the system catchall handler, a thread may have an arbitrary number of each type of handler simultaneously established.

```
\\
ASTs and ULTRIX signal handlers are asynchronous software condition handlers
in this standard.
\\
```

15.4.1 Primary Vectored Handlers

Vectored handlers may only be established at runtime, and are independent of the procedure stack frame structure of the executing thread. Vectored handlers are normally utilized to provide language independent services such as debugging.

When a condition is raised, the system searches for primary vectored handlers. No other types condition handlers are invoked until all primary vectored handlers have been invoked and have reraised the condition.

Primary vectored handlers are invoked in FIFO order with respect to when they were established.

15.4.2 Invocation-Based Handlers

An invocation-based handler is established when a procedure becomes current whose invocation descriptor specifies a condition handler. Thus, invocation-based handlers are usually bound to a procedure at compile time, and are located at run time via the procedure's invocation descriptor.

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

If all primary vectored handlers reraise the condition, then the system searches for invocation-based handlers. The invocation-based handlers which may be invoked are those established by active procedures, from the most current procedure to the oldest predecessor.

15.4.3 Last Chance Vectored Handlers

Like primary vectored handlers, last chance vectored handlers may only be established at runtime, and are independent of the procedure stack frame structure of the executing thread.

If all invocation-based handlers reraise the condition, then the last chance vectored handlers are invoked in LIFO order with respect to when they were established.

15.4.4 System Catchall Handler

Should all other condition handlers reraise the condition, then the system catchall handler is invoked.

The system catchall handler is not established by the 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, or any other system-defined action.

15.5 Establishing Handlers

15.5.1 Establishing Primary Vectored Handlers

A thread may establish a primary vectored handler, to be called in FIFO order after all previously established primary vectored handlers have reraised, via a system supplied function.

Each PRISM system provides concrete language bindings to this function, which is abstractly defined in this standard as ESTABLISH_PRIMARY_HANDLER.

ESTABLISH_PRIMARY_HANDLER accepts two arguments:

- HANDLER: specifies the condition handler to be established as a primary vectored handler.
- [OPTIONAL OUTPUT] ESTABLISHMENT: if this argument is provided, a specifier for this vectored handler establishment will be written to the argument.

ESTABLISH_PRIMARY_HANDLER returns a status value.

15.5.2 Establishing Invocation-based Handlers

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

An invocation descriptor for which FLAGS<HAS_HANDLER> = 1 specifies the self-relative offset of exactly one condition handler in the CONDITION_HANDLER field. The condition 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 no longer established when that descriptor is removed from the invocation chain (that is, when the procedure invocation designated by the descriptor returns or is unwound).

Thus, the set of invocation-based handlers which is established at any moment is defined by the current procedure call structure.

15.5.3 Establishing Last Chance Vectored Handlers

A thread may establish a last chance vectored handler, to be called in LIFO order before any previously established last chance vectored handlers are called, via a system supplied function.

Each PRISM system provides concrete language bindings to this function, which is abstractly defined in this standard as ESTABLISH_LAST_CHANCE_HANDLER.

ESTABLISH_LAST_CHANCE_HANDLER accepts two arguments:

- HANDLER: specifies the condition handler to be established as a last chance vectored handler.
- [OPTIONAL OUTPUT] ESTABLISHMENT: if this argument is provided, a specifier for this vectored handler establishment will be written to the argument.

ESTABLISH_LAST_CHANCE_HANDLER returns a status value.

15.5.4 The System Catchall Handler

The system catchall handler may not be established or modified by any user mode code which conforms to this standard. This condition handler always exists, is always valid, and has system-defined effects.

15.6 Removing Vectored Handlers

A thread may remove a previously established vectored handler by calling a system supplied function.

Each PRISM system provides concrete language bindings to this function, which is abstractly defined in this standard as REMOVE_VECTORED_HANDLER.

REMOVE_VECTORED_HANDLER accepts one argument:

- ESTABLISHMENT: specifies the vectored handler establishment to be removed from the set of established vectored handlers. This value must be one previously returned by ESTABLISH_PRIMARY_HANDLER or ESTABLISH_LAST_CHANCE_HANDLER.

REMOVE_VECTORED_HANDLER returns a status value.

15.7 Arguments Passed to Handlers

Every condition handler is called as a function which returns a status value and is passed two writable arguments:

1. A *condition record* passed by reference.
2. A *mechanism record* passed by rererence.

15.7.1 Condition Record

The condition record is the root of a list of one or more condition records. Each condition record describes one condition, including the condition specifier and the arguments associated with that condition.

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

Each condition record is quadword aligned and is defined as follows.

```
                                                      Quadword Aligned
+---------------+---------------+---------------+---------------+
|                       CONDITION_VALUE                         | :0
+---------------+---------------+---------------+---------------+
|                   CONDITION_VALUE_QUALIFIER                   | :4
+---------------+---------------+---------------+---------------+
|                       CONDITION_FLAGS                         | :8
+---------------+---------------+---------------+---------------+
|                       CONDITION_LIST                          | :12
+---------------+---------------+---------------+---------------+
|                       PROCESSOR_STATUS                        | :16
+---------------+---------------+---------------+---------------+
|                      CONDITION_ADDRESS                        | :20
+---------------+---------------+---------------+---------------+
|                            zero                               | :24
+---------------+---------------+---------------+---------------+
|                       ARGUMENT_COUNT                          | :28
+---------------+---------------+---------------+---------------+
|                                                               | :32
+----                                                       ----+
|                                                               |
+----                CONDITION_ARGUMENT(1)                  ----+
|                                                               |
+----                                                       ----+
|                                                               |
+---------------+---------------+---------------+---------------+
                             .......
+---------------+---------------+---------------+---------------+
|                                                               | :X
+----                                                       ----+
|                                                               |
+----                CONDITION_ARGUMENT(n)                  ----+
|                                                               |
+----                                                       ----+
|                                                               |
+---------------+---------------+---------------+---------------+

  x = 16*(n - 1) + 32
```

CONDITION_VALUE is a longword denoting the condition.

CONDITION_VALUE_QUALIFIER is a longword whose interpretation depends on CONDITION_VALUE (see section "Status Codes and Condition Values").

CONDITION_FLAGS is a set of flag bits. These flag bits are significant only in the primary condition record; they are *undefined* in secondary condition records.

- CONDITION_FLAGS<UNWINDING> = 1 if the condition handler is being invoked because of an unwind operation. This means that the procedure invocation which established this handler is being removed from the invocation chain.
- CONDITION_FLAGS<EXIT_UNWIND> = 1 if the condition handler is being invoked because of an unwind operation that will terminate execution of the thread.

  CONDITION_FLAGS<EXIT_UNWIND> must be 0 unless CONDITION_FLAGS<UNWINDING> = 1.
- CONDITION_FLAGS<NONCONTINUABLE> = 1 if the condition handler must not return STATUS$_CONTINUE.
- CONDITION_FLAGS<DURING_AST> = 1 if an asynchronous software condition handler has resignaled or has been unwound. This indicates that a nested condition was raised or an unwind was initiated during asynchronous condition handling, which was not handled within the scope of the most current active asynchronous condition handler.

CONDITION_LIST is either zero or is the address of the next condition record in the list.

PROCESSOR_STATUS is the processor status stored when the condition was raised.

CONDITION_ADDRESS is the value of the program counter when the condition was raised.

ARGUMENT_COUNT is the number of condition-specific quadword arguments in the condition record.

Each **CONDITION_ARGUMENT(n)** is four longwords which provide additional information specific to the condition, and may contain information intended for display in messages.

15.7.1.1 Condition Arguments

There are two forms of condition arguments:

- immediate condition argument descriptors
- indirect condition argument descriptors

15.7.1.1.1 Immediate Condition Argument Descriptors

Immediate condition argument descriptors contain condition argument values that require eight bytes of storage or less.

```
                                                  Quadword Aligned
+---------------+---------------+---------------+---------------+
|                       ARGUMENT_LONGWORD_1                     | :0
+---------------+---------------+---------------+---------------+
|                       ARGUMENT_LONGWORD_2                     | :4
+---------------+---------------+---------------+---------------+
|   DATA_TYPE   |             zero              |     CLASS     | :8
+---------------+---------------+---------------+---------------+
|                            EXTENT                             | :12
+---------------+---------------+---------------+---------------+
```

CLASS is DESCR$C_IMMEDIATE_ARGUMENT.

DATA_TYPE is the data type of the condition argument.

```
\\
Data types will be defined in a future revision of this standard.
\\
```

ARGUMENT_LONGWORD_1..ARGUMENT_LONGWORD_2 contain the value of the argument. The argument value must be stored starting at byte 0.

EXTENT specifies the number of bytes of contiguous storage occupied by the argument data. The maximum value of EXTENT is 8.

15.7.1.1.2 Indirect Condition Argument Descriptors

Indirect condition argument descriptors specify condition argument values that require more than eight bytes of storage.

```
                                                 Quadword Aligned
+---------------+---------------+---------------+---------------+
|                            EXTENT                             | :0
+---------------+---------------+---------------+---------------+
|                            POINTER                            | :4
+---------------+---------------+---------------+---------------+
|   DATA_TYPE   |             zero              |     CLASS     | :8
+---------------+---------------+---------------+---------------+
|                             zero                              | :12
+---------------+---------------+---------------+---------------+
```

CLASS is DESCR$C_INDIRECT_ARGUMENT.

DATA_TYPE is the data type of the condition argument.

POINTER is the address of the first byte of storage occupied by the data.

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

15.7.1.2 Condition Records for Hardware Conditions

All hardware exceptions have a hardware *exception frame* associated with them. This frame is defined for each exception by the PRISM hardware architecture.

The meaning of the CONDITION_ADDRESS, PROCESSOR_STATUS, and the CONDITION_ARGUMENT fields is determined by the hardware exception frame, which is the source of the information for these condition record fields.

For some hardware exceptions, the CONDITION_ADDRESS specifies the instruction which caused the exception; for other hardware exceptions, CONDITION_ADDRESS specifies some instruction following that which caused the exception. The meaning of CONDITION_ADDRESS is defined for each hardware exception by the hardware architecture, as is the PROCESSOR_STATUS.

In addition to the condition address and processor status, most hardware exceptions have additional exception-specific information defined in the exception frame. It is this additional information which is converted to condition argument format and marshalled into the condition record for the hardware exception. The condition arguments correspond one-for-one with the items defined in the hardware exception frame.

The PRISM hardware architecture should be consulted for further information.

15.7.1.3 Condition Records for FLBC Instruction Faults

The Fault On Low Bit Clear instruction fault is unique among the hardware exceptions in as much as the condition arguments for this condition are *not* the information defined by the PRISM hardware architecture for the exception.

- The hardware architecture defines the first longword in the exception frame to be zero; the first argument in the condition record is the displacement field of the faulting instruction.
- The hardware architecture defines the second longword in the exception frame to be the faulting instruction; the second argument in the condition record is the contents of the register specified by the FLBC register operand.

15.7.1.4 Condition Records for Synchronous Software Conditions

When a software condition is raised, the condition record and one or more condition records must be specified for the first argument to condition handlers invoked for that condition (see section "Raising Software Conditions").

However, the CONDITION_ADDRESS and PROCESSOR_STATUS fields in each condition record *cannot* be specified when the condition is raised. These fields are defined by the system as the virtual address at which the software condition was raised, and the processor status when the condition was raised.

15.7.1.5 Modification of Condition Information by Handlers

The condition record, condition records, and condition arguments are always allocated in writeable memory, and handlers may write to any location in these data structures.

The effect of a handler modifying this condition information is as follows.

1. If CONDITION_FLAGS<NONCONTINUABLE> in the primary condition record is changed from 0 to 1, then the condition handler which made the modification must not return STATUS$_CONTINUE, nor may any handler subsequently invoked for the condition return STATUS$_CONTINUE.
2. If CONDITION_FLAGS in the primary condition record is modified except as specified above, there is no effect after the condition handler completes; all handlers subsequently invoked for the condition receive a primary condition record with the flags unmodified.

   In particular, if a condition handler changes CONDITION_FLAGS<NONCONTINUABLE> from 1 to 0, that handler modification must not return STATUS$_CONTINUE, and any and all handlers subsequently invoked for the condition will be invoked with CONDITION_FLAGS<NONCONTINUABLE> = 1.
3. Except as specified above, all changes made to the condition information will be visible to handlers subsequently invoked for the condition.

4. Any other effects of modifying the condition information are not defined by this standard.

15.7.2 Mechanism Record

The mechanism record describes the environment within which the condition handler is executing.

The mechanism record is quadword aligned and is defined as follows.

```
                                                    Quadword Aligned
+--------------+--------------+--------------+---------------+
|                        STACK_VALID                         | :0
+--------------+--------------+--------------+---------------+
|                       ESTABLISHER_FP                       | :4
+--------------+--------------+--------------+---------------+
|                      RETURN_STATUS_R8                      | :8
+--------------+--------------+--------------+---------------+
|                      RETURN_STATUS_R9                      | :16
+--------------+--------------+--------------+---------------+
```

STACK_VALID is 0 if the system has detected that the thread's stack is corrupt during the search for invocation-based handlers, and 1 if the stack is valid. STACK_VALID will always be 1 when an invocation-based handler is invoked; only vectored handlers are invoked if the stack is corrupt.

ESTABLISHER_FP is the current value of the frame pointer in the procedure invocation which established the handler. ESTABLISHER_FP is meaningful only when an invocation-based handler is invoked. It is *undefined* when a vectored handler is invoked.

RETURN_STATUS_R8 provides a means for condition handlers to determine the contents of register R8 when execution resumes after a handler returns STATUS$_CONTINUE or an unwind operation completes (see "Modification of the Mechanism Record by Handlers").

RETURN_STATUS_R9 provides a means for condition handlers to determine the contents of register R9 when execution resumes after a handler returns STATUS$_CONTINUE or an unwind operation completes.

15.7.2.1 Modification of the Mechanism Record by Handlers

The mechanism record is always allocated in writeable memory, and condition handlers may write to any location in the record.

The effect of a condition handler modifying the mechanism record is as follows.

1. If ESTABLISHER_FP is modified, there is no effect after the condition handler completes; this field is reset by the system before each handler is invoked.
2. If STACK_VALID is modified, there is no effect after the condition handler completes; all handlers subsequently invoked for the condition receive a mechanism record with STACK_VALID unmodified.
3. Changes made to RETURN_STATUS_R8..RETURN_STATUS_R9 will be visible to condition handlers subsequently invoked for this condition.

4. If an unwind operation occurs while a condition is active, and if the optional condition record argument was not specified to the unwind operation, and if one or more active condition handlers are removed from the procedure invocation chain, then the contents of registers R8..R9 when control passes to the unwind target location will be the contents last written to RETURN_STATUS_R8..RETURN_STATUS_R9 for the mechanism record of the most recently raised condition.

   That is, when a condition handler is unwound, and a condition record is not specified, then the last condition handler, if any, which modifies one of these fields before unwinding is initiated will determine the contents of R8..R9 at the unwind target.

5. If a condition handler returns STATUS$_CONTINUE, then the contents of registers R8..R9 when execution resumes at the continue address will be the contents of RETURN_STATUS_R8..RETURN_STATUS_R9 in the mechanism record.

   Therefore the last condition handler, if any, which modifies one of these fields will determine the contents of R8..R9 when execution continues.

6. Any other effects of modifying the mechanism record are not defined by this standard.

15.8 Access to Current Procedure Context

A condition handler may fetch the current context of the procedure invocation within which a condition was raised, and may construct the current context of any invocation on the procedure invocation chain, by calling system supplied functions.

Each PRISM system provides concrete language bindings to these functions, which are abstractly defined in this standard as GET_CONTEXT and GET_PRECEEDING_CONTEXT.

GET_CONTEXT provides the context of the procedure invocation which immediately proceeds the most current active handler on the procedure invocation chain. If that handler was invoked because a condition was raised, this will be the current context of the procedure invocation within which the condition was raised. If that handler was invoked during unwind operation, this will be the current context of the establisher of the handler.

GET_CONTEXT accepts one argument:

- CONTEXT_BUFFER: the address of a 256-byte quadword aligned region of writeable memory into which the procedure context is written.

GET_CONTEXT returns a status value in R8..R9, and provides the procedure context in the memory specified by CONTEXT_BUFFER as follows.

```
                                                Quadword Aligned
+---------------+---------------+---------------+---------------+
|                        PROGRAM_COUNTER                        | :0
+---------------+---------------+---------------+---------------+
|                          REGISTER_SP                          | :4
+---------------+---------------+---------------+---------------+
|                          REGISTER_FP                          | :8
+---------------+---------------+---------------+---------------+
|                          REGISTER_R3                          | :12
+---------------+---------------+---------------+---------------+
                            .......
+---------------+---------------+---------------+---------------+
|                          REGISTER_R61                         | :244
+---------------+---------------+---------------+---------------+
|                          REGISTER_R62                         | :248
+---------------+---------------+---------------+---------------+
|                          REGISTER_R63                         | :252
+---------------+---------------+---------------+---------------+
```

GET_PRECEEDING_CONTEXT takes as input the procedure context provided by a prior invocation of GET_CONTEXT or GET_PRECEEDING_CONTEXT, and returns the current context of the immediately preceeding procedure invocation.

GET_PRECEEDING_CONTEXT accepts one argument:

- CONTEXT_BUFFER: the address of a 256 byte quadword aligned region of writeable memory. When GET_PRECEEDING_CONTEXT is called, the memory specified by CONTEXT_BUFFER must contain the procedure context returned by a prior call to GET_CONTEXT or GET_PRECEEDING_CONTEXT. When this function returns, the memory specified by CONTEXT_BUFFER will contain the context of the procedure invocation preceeding that for which the context was passed as an argument.

GET_PRECEEDING_CONTEXT returns a status value in R8..R9, and provides the preceeding procedure context in the same form at that provided by GET_CONTEXT.

15.9 Handler Completion and Return Value

When a condition handler has finished all its processing, it must complete its execution by *reraising* the condition, *continuing* execution of the thread at the location at which the condition was raised, or by initiating procedure invocation *unwinding*.

15.9.1 Reraise

If a condition handler determines that additional handlers should be invoked for the condition (because it could not handle the condition), it can reraise the condition by setting register R8 to STATUS$_RERAISE and performing a normal return.

Reraise causes the next condition handler to be invoked (see section "Order of Handler Invocation"). This next handler is invoked with an updated mechanism record and with the condition record as it was left by the handler which reraised the condition.

If all condition handlers established by the thread reraise the condition, the system catchall handler will be invoked, with system-defined results.

15.9.2 Continue

A condition handler can continue execution of the thread at the location at which the condition was raised by setting register R8 to STATUS$_CONTINUE and performing a normal return.

This will cause execution to resume at the condition address, with the context of the interrupted procedure restored, *except* registers R8..R9. Registers R9..R9 are restored from RETURN_STATUS_R8..RETURN_STATUS_R9 in the mechanism record, so modification made to these fields by a condition handler will be reflected in the context of the interrupted procedure when its execution resumes.

If STATUS$_CONTINUE is returned and CONDITION_FLAGS<NONCONTINUABLE> = 0, then the register context of the procedure invocation in which the condition was raised is restored to its original state (except R8..R9), and execution continues at the condition address. The contents of registers R8..R9 when execution continues will be will be the contents of RETURN_STATUS_R8..RETURN_STATUS_R9 in the mechanism record.

If STATUS$_CONTINUE is returned and CONDITION_FLAGS<NONCONTINUABLE> = 1, then a nested condition is raised with CONDITION_VALUE = STATUS$_NONCONTINUE_CONDITION, indicating that an attempt was made to continue from a noncontinuable condition. This second condition is also noncontinuable.

15.9.2.1 Restrictions on Continuation

The PRISM 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 most hardware exceptions is possible, but there are some restrictions.

Arithmetic traps cannot be restarted, since they are instruction aborts and not all information is stored for a restart. The only way in which software may guarantee the ability to continue from this type of exception is by placing DRAIN instructions around an exception site.

```
\\
Although this allows continuation, it is neither practical nor desirable
because of the negative effects on application performance.
\\
```

User stack alignment abort also does not save enough information to allow a restart or continuation.

Software conditions 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 condition. However, since compiler-generated code at any instant typically relies on previous error free execution, continuing from a software condition may result in unpredictable behavior unless the handler has explicitly fixed the cause of the condition in such a way as to be transparent to subsequent code.

15.9.3 Unwind

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 condition 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 will remove from the procedure invocation chain all invocations up to the target, and continue thread execution at the specified location in that procedure.

Before control is transfered to the target location, the unwind operation invokes each invocation-based handler which was established by any procedure invocations being removed. These handlers are invoked with a condition record indicating that an unwind is in progress, and a mechanism record describing the environment from which the unwind was initiated. This allows each procedure invocation being discarded to perform clean-up processing before its context is lost.

Once this phase has completed, the target invocation's register saved context is restored and the execution is continued at the specified location. In the case that a condition handler is unwound, R8..R9 are restored from the mechanism record, allowing a status to be returned to the target of the unwind.

One effect of the unwind operation is to discard the condition handler invocation. Control will never be returned to the point at which the unwind was initiated, and that handler invocation can therefore never return to its caller.

The details of unwinding are discussed in section "Procedure Invocation Unwinding".

15.9.3.1 Exit Unwind

Since processes on the PRISM system are multithreaded, it is necessary for a thread which is terminating execution to clean up its use of shared process resources such as the virtual address space.

Because of this, user mode thread exit may accomplished *only* by unwinding. A special form of unwind, referred to as *exit unwind*, invokes all established invocation-based handlers with a condition record specifying that an exit unwind is in progress, removes all procedure invocations up to the beginning of the call heirarchy, and terminates execution of the thread.

15.10 Order of Handler Invocation

When a condition occurs, established condition handlers are invoked in a specific order.

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

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

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

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

15.10.1 Nested Conditions

A nested condition occurs if a condition is raised while a condition handler is active.

When a nested condition 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 condition 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 that in which the nested condition was raised (item 1) if there are zero invocations in item 2.
4. The procedure invocation within which the active condition that immediately preceeded the nested condition was raised; that is, the invocation in which the condition 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 condition that immediately preceeded the nested condition, and all of which resignaled.
6. The establisher of the most current handler.

   This is the same as the invocation in which the condition that immediately preceeded the nested condition was raised (item 4) 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 will be invoked in reverse order with respect to that in which their establishers were invoked, as defined above.

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

Any handlers established by the invocations described by items 4, 5, and 6 will be invoked if, and *only* if, the invocation 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 condition 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 conditions occur, this procedure invocation chain structure is repeated for those further nested conditions, and invocation-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 condition 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.

15.10.2 Steps of Searching for and Invoking Handlers

When a condition is raised, the steps that implement the above capabilities are as follows.

1. If an alternate condition stack has been established, switch to the alternate condition stack (see section "Alternate Condition Stack for Vectored Handlers").
2. Locate the first-established primary vectored handler, if any.
3. If no established primary vectored handler was located, go to step 8.
4. Invoke the vectored handler just located.
5. If the handler returns STATUS$_CONTINUE or initiates an unwind, switch back to the primary stack and exit these steps.
6. Locate the next-established primary vectored handler, if any.
7. Go to step 3.
8. If an alternate condition stack has been established, switch back to the primary stack.
9. Let current_invocation be the procedure invocation in which the condition was raised.
10. If current_invocation does not establish a handler, go to step 21.
11. Invoke the handler established by current_invocation.
12. If the handler returns STATUS$_CONTINUE or initiates an unwind, exit these steps.
13. If current_invocation is not itself an active handler, go to step 21.
14. Locate the establisher of current_invocation.
15. Let current_invocation be the procedure invocation which invoked current_invocation.
16. If the current_invocation does not establish a handler, go to step 20.
17. If the handler is not flagged as reinvokable by its establisher, go to step 20.
18. Invoke the handler established by current_invocation.
19. If the handler returns STATUS$_CONTINUE or initiates an unwind, exit these steps.
20. If current_invocation is not the establisher located in step 14, go to step 15.
21. If current_invocation is the beginning of the procedure invocation chain, go to step 24.
22. Let current_invocation be the procedure invocation which invoked current_invocation.
23. Go to step 10.
24. If an alternate condition stack has been established, switch to the alternate condition stack.
25. Locate the last-established last chance vectored handler, if any.
26. If no established last chance vectored handler was located, go to step 31.
27. Invoke the vectored handler just located.
28. If the handler returns STATUS$_CONTINUE or initiates an unwind, switch back to the primary stack and exit these steps.
29. Locate the previously-established last chance vectored handler, if any.
30. Go to step 26.
31. Invoke the system catchall handler.

32. Force thread to terminate execution by initiating an exit unwind.

15.11 Other Properties of Condition Handlers

15.11.1 Access to Memory

Conditions can be raised when the current value of one or more variables is in registers rather than in memory.

Because of this, a condition handler, and any descendant procedure called directly or indirectly by a handler, must in general not access any memory except arguments explicitly passed to the procedure and memory that the procedure allocated.

This rule can be violated for specific memory locations only by agreement between the handler and all procedures which might access those memory locations, and such a handler is not standard conforming.

15.11.2 Alternate Condition Stack for Vectored Handlers

To avoid having vectored handlers utilize the main procedure stack, a thread may specify an alternate stack for invocation and use of vectored handlers. This is done by calling a system supplied function.

Each PRISM system provides concrete language bindings to this function, which is abstractly defined in this standard as CREATE_CONDITION_STACK.

CREATE_CONDITION_STACK accepts one argument:

- STACK_SIZE: the size in bytes for the alternate condition stack.

CREATE_CONDITION_STACK will always allocate a minimum size for the alternate condition stack, generally the size required to raise the largest architecture-defined hardware condition and a stack overflow condition. If STACK_SIZE is less than this system-defined minimum size, then the minimum stack size will be allocated.

CREATE_CONDITION_STACK returns a status value, which may describe an error condition if, for example, the alternate condition stack could not be allocated or if an alternate condition stack was already established.

If a condition stack is defined for a thread, it is used to invoke *all* user mode vectored handlers. Thus, mechanism and condition records are first delivered to the primary vectored handlers on the condition stack. If all of these handlers resignal, the mechanism and condition records are transferred to the thread's main stack and any invocation-based handlers are invoked. If all of these resignal, or the main stack is corrupted, the condition and mechanism records are transferred back to the condition stack and the last chance handlers are invoked.

Once created, the alternate condition stack is permanent with respect to the thread; it cannot be disabled or removed.

15.11.3 Invalid Thread Stack

If, during the search for and invocation of invocation-based handlers, the system detects that the thread's primary stack or alternate condition stack is corrupt, then the following steps take place.

1. STACK_VALID in the mechanism record is set to 0.
2. The search for handlers immediately proceeds to the last chance vectored handlers (step 24 in the previous section).
3. If all last chance vectored handlers resignal, then the system catchall handler is invoked.

16 Procedure Invocation Unwinding

Unwinding transfers control in a thread from the location at which the unwind operation is initiated to the *target location* in a *target invocation*.

This results in removal from the procedure invocation chain of all procedure invocations, including the invocation in which the unwind request was initiated, up to the target procedure invocation, after which thread execution continues at the target location. Once an unwind is initiated, control never returns to the point at which the unwind was initiated.

Unwinding does not require a condition handler to be active; it may be used by languages to implement nonlocal GOTO.

Before control is transfered to the target location, the unwind operation invokes each invocation-based handler which was established by any procedure invocations being removed. These handlers are invoked with a condition record indicating that an unwind is in progress, and a mechanism record describing the environment from which the unwind was initiated. This allows each procedure invocation being discarded to perform clean-up processing before its context is lost.

Once this phase has completed, the target invocation's saved register context is restored and the execution is continued at the specified location.

16.1 Types of Unwind

There are three types of unwind requests:

1. Nonlocal GOTO

   Nonlocal GOTO transfers control to a specified location in a specified procedure invocation.

   The target procedure invocation is specified by the address of its stack frame. Thus, a procedure invocation with a register frame may *not* be the target of a nonlocal GOTO.

   If the target location is not specified, then the target location is the current return address in the target invocation.

   This type of unwind may be initiated from any context: from any type of handler, and when no handler is active.

2. Caller of establisher

   Unwind to caller of establisher transfers control to the caller of the establisher of the most current active invocation-based condition handler. The target location in the caller of the establisher may be specified; if not specified, the target location is the current return address in that procedure invocation.

   A procedure with a register frame may *not* be the target of an unwind to caller of establisher.

   This type of unwind may only be initiated by an active invocation-based handler, or by a procedure called directly or indirectly from such a handler.

3. Exit unwind

   Exit unwind removes every procedure invocation in the invocation chain, after which execution of the thread is terminated.

This type of unwind may be initiated from any context: from any type of handler, and when no handler is active.

Since processes on the PRISM system are multithreaded, it is necessary for a thread which is terminating execution to clean up its use of shared process resources such as the virtual address space. Because of this, user mode thread exit may accomplished *only* by initiating an exit unwind.

16.2 Unwind Initiation

A thread may initiate an unwind operation by calling a system supplied function.

Each PRISM system provides concrete language bindings to this function, which is abstractly defined in this standard as UNWIND.

UNWIND returns a status value, although it only returns to the caller if an error is detected in the argument list.

UNWIND accepts six arguments:

- FRAME: Zero or the address of the stack frame of the target invocation.
- CALLER_OF_ESTABLISHER: TRUE or FALSE
- EXIT_UNWIND: TRUE or FALSE
- TARGET_PC: Zero or the address of the target location.
- CONDITION_RECORD: Zero or the address of a condition record
- COLLAPSE_STACK: TRUE or FALSE

FRAME, if non-zero, specifies the address of the stack frame of the target procedure invocation to be unwound to.

CALLER_OF_ESTABLISHER = TRUE specifies unwind to the caller of the establisher of the most current active invocation-based condition handler. CALLER_OF_ESTABLISHER must be not be TRUE unless a invocation-based handler is active; otherwise, an error condition is raised.

EXIT_UNWIND = TRUE specifies an exit unwind.

These first three arguments are the means of specifying one of the three mutually exclusive types of unwind. The error STATUS$_INVALID_ARGUMENTS is returned if more than one of the types of unwind are specified, or if none of the types of unwind are specified.

TARGET_PC specifies the address within the target invocation at which to continue execution. It may be used with either the FRAME or the CALLER_OF_ESTABLISHER arguments. If TARGET_PC is zero for these types of unwind, the current return address in the target procedure invocation is used. If TARGET_PC is non-zero and EXIT_UNWIND is TRUE, STATUS$_INVALID_ARGUMENTS is returned.

CONDITION_RECORD may be used to specify an optional condition record. If specified, this argument is used as the condition record passed to each handler called during the unwind operation. The unwind operation checks the validity of the condition record; STATUS$_INVALID_ARGUMENTS is returned if an ill-formed condition record is specified. If no condition record is specified, then the system allocates a default condition record.

COLLAPSE_STACK is used to determine whether, before execution resumes at the target location, the stack is collapsed back to the point at which the target procedure invocation last left it, or left as it was when the unwind operation was initiated. If COLLAPSE_STACK is FALSE, a procedure which initiates an unwind may return data to the target invocation by leaving it on the top of the stack.

16.3 Arguments Passed to Handlers

The arguments passed to handlers invoked by the unwind operation are the same as those passed to handlers for all other conditions: a condition record and a mechanism record.

These records have certain properties specific to the unwind operation.

16.3.1 Condition Record

When an unwind operation is in progress, CONDITION_FLAGS<UNWINDING> = 1 in the primary condition record. If the unwind is an exit unwind operation, then in addition CONDITION_FLAGS<EXIT_UNWIND> = 1 in the primary condition record.

If the CONDITION_RECORD argument is specified when the unwind is initiated, then all other properties of the condition record are determined by CONDITION_RECORD.

If CONDITION_RECORD argument is not specified, then a default condition record is supplied which specifies exactly one condition record in which CONDITION_VALUE = STATUS$_UNWINDING.

16.4 Order of Handler Invocation

When an unwind operation takes place, all invocation-based condition handlers are invoked which were established by any procedure invocation being removed from the invocation chain. These handlers are invoked in reverse order with respect to that in which they were established.

Since vectored handlers and the system catchall handler are not associated with a procedure invocation, these handlers are never invoked during an unwind (although they will be invoked if a condition is raised during the unwind operation).

16.4.1 Multiply Active Unwind Operations

During an unwind operation, another unwind operation may be initated. This may occur, for example, if a handler which is invoked for the original unwind initiates another unwind, or if a condition is raised in the context of such a handler and a handler invoked for that condition initiates another unwind operation.

An unwind which is initiated while a previous unwind is active is either a *nested unwind* or an *overlapped unwind.*

16.4.1.1 Nested Unwind

A nested unwind is an unwind to caller of establisher or a nonlocal goto 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 remove any procedure invocation which would have been removed 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 will be complete and the previous unwind will once again be the most current unwind operation.

16.4.1.2 Overlapping Unwind

An overlapping unwind is an exit unwind, an unwind to caller of establisher, or a nonlocal goto 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 removes one or more procedure invocations that would have been removed by the previously active unwind.

An overlapping unwind is detected immediately after the most current active unwind handler is removed from the procedure invocation chain. 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.

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 removed from the invocation chain.

The target location of the merged unwind operation is the target location associated with the oldest preceseccor 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 ignored.

The condition and mechanism arguments of the merged unwind operation are those associated with the overlapping unwind; the arguments associated with the previously active unwind are discarded.

After the colliding unwinds are merged, the unwind operation continues from the point of the collision.

These rules for merging overlapping unwinds and continuing do *not* apply when an overlapping unwind to caller of establisher or an overlapping nonlocal GOTO collide with a previously active exit unwind. If an overlapping non-exit unwind collides with a previously active exit unwind, the noncontinuable STATUS$_COLLIDED_EXIT_UNWIND condition is raised.

```
\\
NOTE TO CALLING STANDARD WORKING GROUP:

I have not yet mechanized overlapping unwind in the steps below.

The overlapping unwind rules will be mechanized before the next public
distribution of the standard.
\\
```

16.4.2 Steps of Searching for and Invoking Handlers

When an unwind operation it initiated, the steps that implement the above capabilities are as follows.

1. If unwind arguments are invalid, return error STATUS$_INVALID_ARGUMENTS.
2. If CONDITION_RECORD specified, let unwind_status = CONDITION_VALUE.
3. If CONDITION_RECORD not specified, supply default condition record and let unwind_status = STATUS$_CONDITION_NORMAL.
4. If CONDITION_RECORD is invalid, return error STATUS$_INVALID_ARGUMENTS.
5. Let current_invocation be the procedure invocation which initiated the unwind operation.
6. If current_invocation = FRAME, go to step 25.
7. If current_invocation is an active handler, go to step 13.
8. If current_invocation establishes a handler, invoke that handler.
9. Remove current_invocation from the procedure invocation chain.
10. If the procedure invocation chain is empty, go to step 28.
11. Let current_invocation be the most current invocation in the procedure invocation chain.
12. Go to step 6.
13. If CONDITION_RECORD not specified in unwind argument list, let unwind_status = RETURN_STATUS_R8..RETURN_STATUS_R9 in the mechanism record which was the second argument to current_invocation.
14. If unwind type not CALLER_OF_ESTABLISHER, go to step 20.
15. If current_invocation is a vectored handler, raise STATUS$_UNWIND_THROUGH_VECTOR condition.
16. Locate caller of establisher of current_invocation.
17. If caller of establisher of current_invocation has register frame, raise STATUS$_INVALID_CONDITION_DESC condition.
18. Let FRAME = address of stack frame of caller of establisher of current_invocation.
19. If current_invocation = FRAME, go to step 25.
20. If current_invocation establishes a handler, invoke that handler.
21. Remove current_invocation from the procedure invocation chain.
22. If the procedure invocation chain is empty, go to step 28.
23. Let current_invocation be the most current invocation in the procedure invocation chain.
24. Go to step 20.

25. Restore saved context of current_invocation. Let R8..R9 = unwind_status. Set stack pointer as specified by COLLAPSE_STACK. Let return_pc = return address in current_invocation.
26. If TARGET_PC specified, let return_pc = TARGET_PC.
27. Exit these steps, resuming execution at TARGET_PC.
28. If unwind type not EXIT_UNWIND, raise STATUS$_TARGET_FRAME_NOT_FOUND condition.
29. Terminate execution of thread.

```
\\
These steps do not incorporate overlapping unwind support.
\\
```

16.4.3 Invalid Thread Stack

If, during the search for and invocation of invocation-based handlers, the system detects that the thread's primary or condition stack is corrupt, the STATUS$_STACK_INVALID condition is raised, interrupting the unwind operation.

This will result in a search for vectored handlers to be invoked for the stack invalid condition. If no vectored handler continues execution of the unwind operation, then the system catchall handler will be invoked.

16.5 Unwind Completion

When a nonlocal GOTO or an unwind to caller of establisher completes, the following properties apply.

- The target procedure invocation is the most current invocation in the procedure invocation chain.
- The saved register context of the target invocation is restored to its state when the invocation was last current, except for registers R8..R9 (even if saved).
- The contents of R8..R9 are determined as follows:
  1. If CONDITION_RECORD was specified to the unwind operation, then R8..R9 contains the CONDITION_VALUE..CONDITION_VALUE_QUALIFIER specified in the condition record.
  2. If CONDITION_RECORD was not specified, and if the unwind operation was initiated while a condition was active, and if one or more active condition handlers were removed from the invocation chain, then the contents of R8..R9 are the contents last written to RETURN_STATUS_R8..RETURN_STATUS_R9 in the mechanism record of the most recently raised condition.
  3. Otherwise R8..R9 contain STATUS$_CONDITION_NORMAL.
- If COLLAPSE_STACK was specified to the unwind operation, then the stack pointer is restored to its state when the invocation was last current; otherwise, the stack pointer is in the state it had the unwind operation was initiated. (Note that if the stack is not collapsed, then the target location must be prepared to deal with this).

- Execution continues at the target location, which defaults to the return PC if not specified.

17 Asynchronous Software Conditions

\\
The next revision of this standard will define interfaces for enabling and disabling asynchronous software conditions, for raising and handling such conditions, the effects of such conditions, and the interaction between such conditions and the rest of the condition handling environment.
\\

17.1 Raising Asynchronous Software Conditions

\\
This will be defined in the next revision of this standard.
\\

17.2 Enabling and Disabling Delivery of Asynchronous Conditions

\\
This will be via the SWASTEN instruction and an abstract functional interface, which will be defined in the next revision of this standard.
\\

17.3 Invocation of Asynchronous Handlers

\\
This will be defined in the next revision of this standard.
\\

17.4 Effect of Conditions Raised in Asynchronous Handlers

\\
This will be defined in the next revision of this standard.

Since asynchronous software conditions can be raised and asynchronous handlers invoked in a procedure context that has little or nothing to do with the asynchronous condition, established invocation-based handlers may not be prepared to deal with conditions that are raised in the context of an active asynchronous handler. Asynchronous handlers have to avoid propagating inapproprite conditions to the invocation-based handlers. Note also that unwinds can propagate out of an asynchronous handler.
\\

17.5 Asynchronous Handler Completion

\\
This will be defined in the next revision of this standard.
\\

18 Interprocedural Synchronization

18.1 Exception Synchronization

The PRISM 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.

When a procedure with a condition handler is called, the entry code sequence must execute a DRAIN instruction before setting FP in order to ensure that pending exceptions are raised in the condition handling context of the calling procedure.

Likewise, when a procedure with a condition handler returns, the return code sequence must execute a DRAIN instruction before resetting FP in order to ensure that pending exceptions are raised in the condition handling context of the returning procedure.

This rule ensures that exceptions are detected in the context within which condition handlers may have been set up to handle such exceptions.

This rule does *not* ensure that exceptions are detected while the procedure within which the exception-causing instruction was issued is current. For example, if a procedure without a condition handler is called, 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 condition handling information is the frame which was current when the exception was detected, *not* the frame which was current when the exception-causing instruction was issued.

If a procedure wants to make sure all exceptions it might cause are detected while it is the current procedure, then a DRAIN must be executed before every procedure call it makes and before it returns.

18.2 Memory Synchronization

The PRISM hardware architecture allows vector memory operations to be executed without automatic hardware synchronization of those vector memory operations with scalar memory operations or with other vector memory operations.

This requires execution of DRAINM instruction to synchronize vector memory operations with other memory operations (vector or scalar) that may reference the same quadword. DRAINM must be executed between a vector memory operation and every other preceeding or following memory operation, unless it can be determined that the vector operation does not potentially conflict with the other memory operations.

The term *unsafe quadword* denotes any quadword that can potentially be referenced by more than one memory operation, where *neither* of the following is true:

1. All references to the quadword are read-only references.
2. All references to the quadword are by scalar operations.

The term *unsafe operation* denotes any reference to an unsafe quadword by any vector load/store instruction. (Only vector memory operations are termed unsafe, since only vector memory operations are potentially unsynchronized with any other memory operations).

Since standard procedure calls do not assume global memory reference agreements across the call interface, all vector memory operations referencing memory potentially visible to any other procedure (for example, global variables, passed arguments, up level addressable storage, or stack temporary storage that may be used by a calling procedure after return) are unsafe operations in a standard call.

Any procedure containing an unsafe operation must adhere to the following conventions.

1. A DRAINM instruction must be executed by a called procedure (typically in the entry code sequence) before the first unsafe operation, to synchronize with memory operations issued before the procedure was called.
2. A DRAINM instruction must be executed by a called procedure (typically in the return code sequence) after the last unsafe operation, to synchronize with memory operations issued after the procedure returns.
3. A DRAINM must be executed by a calling procedure between each unsafe operation and the next standard call, to synchronize with memory operations subsequently issued by called procedures.
4. A DRAINM must be executed by a calling procedure between each standard call and the next unsafe operation, to synchronize with memory operations previously issued by called procedures.

That is, any procedure that issues vector loads or stores is responsible to synchronize with potentially conflicting memory operations by any other procedure. (This may result in execution of redundant DRAINM instructions, which is a consequence of synchronizing with scalar load/stores across standard calls.)

```
\\
Adherence to the above conventions is not required when it can be determined
(by compiler analysis or by agreements across procedure call interfaces) that a
procedure contains no unsafe operations relative to calling and called
procedures.

This is, however, not a standard call.
\\
```

19 User Mode Thread Architecture and Conventions

The PRISM operating systems provide multiple threads of execution within a process. A *thread* is the entity that is scheduled for execution on a processor. A *process* includes an address space and at least one thread of execution.

This standard applies only to threads which execute within the context of a user process in user mode and are scheduled according to software priority. All subsequent uses of the term *thread* in this standard refer to such user mode process threads only.

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 processor status, program counter, stack pointer, scalar registers R2..R63, and vector registers V0..V15.

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.

Threads may create and delete other threads, and may affect other threads via mechanisms provided by the operating system. However, the operating system does not define the relationship of threads within a process to one another; there is no heirarchical arrangement of threads. Neither does the operating system dictate policy for use of process resources by individual threads within the process.

The *user mode thread architecture* defined by this calling standard provides the additional properties and conventions to allow multiple threads to coordinate multilanguage execution within a process, including

- the relationship to one another of threads within a process
- synchronization between multiple threads in a process
- thread management of process resources
- the interaction of threads via asyncronous events

19.1 Goals

- Simplicity

  Support a set of primitives that are adequate for use by languages and utilities to implement multithreaded applications and tools, and which can be easily understood and documented.

- Support multiple languages and facilities

  The user mode thread architecture must ensure that threads and sets of threads can meet all important requirements of PRISM languages and software products.

  These requirements include (but are not limited to):

  - Closely-coupled threads—a set of threads executing the same code sequence, such as different iterations of a loop, with fine grained synchronization between threads.
  - Tasking threads—a thread that runs a high level language procedure as a separate task, with language-defined synchronization between tasks.
  - Work queue threads—a set of threads that maintain a work queue, inserting and removing work items in cooperation with one another.

  - LIB$ threads—direct use of the PRISM operating system thread management services from high level languages, via run time library interfaces.

- Avoid restricting concurrent execution

  The user mode thread architecture must support concurrent execution in a single process of multiple threads and sets of threads under the control of different languages and facilities.

- Avoid restricting multilanguage execution within a thread

  The user mode thread architecture must allow code compiled from multiple Digital languages to execute correctly within a single thread.

- Support low-overhead multithread execution

  The user mode thread architecture must support multithread execution that is efficient enough to allow fined grained parallelism.

- Support thread management of process resources

  The user mode thread architecture must define mechanisms to allow multiple threads to cooperatively allocate and manage shared process resources (such as virtual memory and other threads). Furthermore, it must define mechanisms to ensure that, when that thread terminates, it is able to free process resources that it allocated.

- Support coordination of conditions between threads

  The user mode thread architecture must support coordination between a cooperating set of threads when a condition occurs in one thread, one thread terminates abnormally, etc. (for example, a collection of cooperating FORTRAN decomposition threads, or a related set of ADA tasks implemented as threads).

- Support thread synchronization

  The user mode thread architecture must define mechanisms that allow multiple threads to synchronize their execution as necessary, while working correctly in a multilanguage environment (for example, FORTRAN threads synchronizing with one another, while ADA tasks are concurrently executing within the same process).

- Support run time programmer tools

  The user mode thread architecture must provide the features necessary for run time PRISM programming tools, such as DEBUG and PCA, to support multiple threads of execution.

- Coexist with multiple system interfaces

  The user mode thread architecture must support and work correctly with the ULTRIX and MICA operating system interfaces.

- Layer cleanly on the PRISM operating system thread management services

  The user mode thread architecture should avoid placing unnecessary requirements on the operating system and should avoid duplicating operating system functions.

- Common Multithread Architecture support

  The user mode thread architecture must support and coexist with the facilities defined by the Common Multithread Architecture.

19.2 Thread Environment Block

The *thread environment block* (TEB) is a data structure associated with each thread. The TEB is used to maintain user mode software context specific to that thread, and to manage resources used by the thread.

The address of a thread's environment block is specified by the contents of the thread's TEB_BASE register (which is always R3). The TEB_BASE register must specify the address of a valid thread environment block at all times, and the TEB must be quadword aligned.

The following figure illustrates the layout of the thread environment block.

```
                                                      Quadword Aligned
+----------------------------------------------------------------+
|                         TEB_LENGTH                             | :0
+----------------------------------------------------------------+
|                         TCR_ADDRESS                            | :4
+----------------------------------------------------------------+
|                         TEB_VM_ZONE                            | :8
+----------------------------------------------------------------+
|                      TLS_ARRAY_ADDRESS                         | :12
+----------------------------------------------------------------+
|                        TLS_ARRAY_FREE                          | :16
+----------------------------------------------------------------+
```

TEB_LENGTH contains the length in bytes of the thread environment block. This 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.

TCR_ADDRESS contains the address of the thread control region for this thread.

TEB_VM_ZONE is the identification of a zone of virtual memory which is unique to the thread and which is autimatically deallocated by the run time system when the thread terminates.

TLS_ARRAY_ADDRESS contains the quadword aligned address of the thread local static storage control array. Use of this field and of the thread local static storage control array is defined in section "Thread Local Static Storage".

TLS_ARRAY_FREE specifies the byte offset of the first unused position in the thread local static storage control array. Use of this field is defined in section "Thread Local Static Storage".

19.3 Thread Control Region

The *thread control region* (TCR) is a data structure associated with each thread, and used to maintain software context specific to that thread. The contents of the TCR are maintained by the operating system on behalf of the thread, and are read-only to user mode.

The thread control region is accessed through a pointer maintained in the thread environment block.

The following figure illustrates the layout of the thread control region.

```
                                              Quadword Aligned
+-------------------------------------------------------------+
|                                                             | :0
|                        Thread ID                            |
|                                                             |
+-------------------------------------------------------------+
|             thread invocation descriptor address            | :12
+-------------------------------------------------------------+
|                   primary stack initial SP                  | :16
+-------------------------------------------------------------+
|                     primary stack limit                     | :20
+-------------------------------------------------------------+
|                     primary stack base                      | :24
+-------------------------------------------------------------+
|                  condition stack initial SP                 | :28
+-------------------------------------------------------------+
|                    condition stack limit                    | :32
+-------------------------------------------------------------+
|                    condition stack base                     | :36
+-------------------------------------------------------------+

******************* THIS IS NOT ACCURATE OR UP TO DATE *****************
****** IT WILL BE UPDATED WHEN THE NEW MICA CHAPTER IS AVAILABLE *******
```

Thread ID contains the operating system supplied ID for this thread.

Thread invocation descriptor address contains the address of the invocation descriptor which designates the entry point of the procedure initially invoked when the thread is created.

Primary stack initial SP contains the initial value of the thread's primary stack pointer.

Primary stack limit contains the lowest address currently allocated to the thread's primary stack. Since the stack grows from high addresses to low addresses, the primary stack locations in use at any instant are those between the primary stack base and the current primary stack pointer.

Primary stack base contains the lowest address allocated to the thread's primary stack. The primary stack locations not in use at any time are those between the current primary stack pointer - 1 and the primary stack limit.

Condition stack initial SP contains the initial value of the thread's condition stack pointer.

Condition stack limit contains the lowest address currently allocated to the thread's alternate condition stack. Since the stack grows from high addresses to low addresses, the condition stack locations in use at any instant are those between the condition stack base and the current condition stack pointer.

Condition stack base contains the lowest address allocated to the thread's alternate condition stack. The condition stack locations not in use at any time are those between the current condition stack pointer - 1 and the condition stack limit.

19.4 Thread Creation and Initialization

```
\\
TBS -- we will specify here the environment and state that applies compatibly
across MICA and ULTRIX when non-system user mode code gets control in a newly
created thread on a system that supports multithreaded processes.

In this context, part of the common RTL is probably considered "system" code,
and we will include the effects of RTL thread initialization in this section.

This will be incorporated when the necessary design description from the MICA
group is available.
\\
```

19.5 Thread Exit

When a thread exits, various actions must be performed. Such actions may include

- Terminate threads created as computational resources specifically for this thread
- Free all thread local static regions
- Free the default VM zone
- Free other resources local to this 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* standard means for initiating thread exit is to initiate an exit unwind operation.

This unwind condition will terminate the thread after all procedures have been unwound.

As the procedures in a thread are unwound, they must respond to the thread exit unwind by performing the appropriate exit actions.

See section "Procedure Invocation Unwinding" for the complete definition of the exit unwind operation.

20 Stack Limit Checking

A program that is otherwise correct can fail because of stack overrun. Stack overrun occurs when extension of the stack (by decrementing SP) allocates addresses not reserved for the current thread's stack. If not explicitly detected in some way, this condition can result in unpredictable behavior because the current thread, writing into what it considers to be stack storage, modifies data allocated to that storage for some other purpose.

The standard conventions for PRISM procedures include checking for stack overflow based on a guard region (a page or contiguous set of pages). The virtual address space allocated for a thread's stack contains one or more null-access pages at the stack's limit (low address end). The total size of this guard region is at least 2*GPLIM bytes, where the value GPLIM is defined to be 4096 bytes by this standard.

```
\\
Threads will probably be created with two contiguous 8K byte (2*GPLIM = 8K)
guard regions, in order to allow handling of stack overflow exceptions. When a
stack overflow occurs, the highest-addressed 8K region will be unprotected to
allow delivery and processing of the stack overflow.  The lowest-addressed 8K
guard region continues to provide the required protection during stack overflow
handling.
\\
```

Most stack limit checking is implicit, based on the fact that any reference to a guard page will cause an access violation fault.

Compilers must generate code such that the stack is not extended past the guard pages into valid storage that is not allocated to the current thread's stack. This requires that the difference between the value of SP and the address of some stack location known to be valid never exceeds GPLIM. Therefore, stack extension can be done by sequences such as:

```
STL   R0, -GPLIM(SP)   ; Touch within GPLIM of the current stack top.
                       ; The stack is still valid for delivery of an exception
                       ; if one occurs
LDA   ext(SP), SP      ; Extend by no more than 2*GPLIM.
                       ; If |ext| ≤  GPLIM then ext-8(SP) may be in a guard page
                       ; and a subsequent small stack extension may cause
                       ; an exception.
                       ; If GPLIM < |ext| ≤  2*GPLIM then ext(SP) may be in a guard
                       ; page and the next stack reference may cause an exception.
```

If the location at 2*GPLIM(SP) is not in a valid stack page belonging to this thread, then the location referenced in the first instruction will be in a guard page and will cause an access violation. If the referenced location is valid, then any new value of SP (within 2*GPLIM of the old value of SP) will not be past the guard pages.

If the stack is being extended by a dynamic amount (value not known at compile time), the new SP value must be checked against the actual stack limit, which is kept in a data structure accessed via R3, the TEB_BASE register. Stack extension by a dynamic amount can be done by sequences such as:

```
                                  ; Assume that amount of additional stack needed is in R30
    LDL    TCR_ADDRESS(R3), R31   ; Get address of thread control region
    ADD    SP, R30, R30           ; Develop address of new SP
    LDL    STACK_LIMIT(R31), R31  ; Get current stack limit
    CMPGE  R30, R31, R31          ; Is new SP ≥  stack limit?
    BLBC   R31, STACK_OVERFLOW    ; If R31 = 0 then new SP < stack limit, so error.
    OR     R0, R30, SP            ; Extend SP by needed amount
    \\
    Note -- STACK_LIMIT is not a very good name for a field that has a defined name
    in the TCR.  When the updated process chapter is available, update this name
    to reflect the real name of the current stack limit in the TCR.
    \\
```

If a stack overflow occurs, the system may extend the thread's stack and reset STACK_LIMIT appropriately. In particular, a stack overflow that occurs in a called procedure might cause the stack to be extended and STACK_LIMIT to be modified in the TCR.

Because of this the STACK_LIMIT value in the TCR must be considered volatile and potentially modified by external procedure calls.

21 Reserved Virtual Memory

Two regions of virtual memory must not be accessed by any user mode code:

- First 64K—virtual addresses 0..65535.
- Last 64K—virtual addresses $2^{32} - 65536..2^{32} - 1$.

These memory regions are reserved, must not be accessed for read, write, or execute, and will normally be protected against any user mode access.

22 Scope and Applicability

This section summarizes the properties that the PRISM Extended Calling Standard requires of user mode code, in contrast to techniques, recommendations, and examples provided by this standard that are not required properties. This section also explains the scope for which these properties are required.

This standard applies specifically to the user mode run time environments of both the PRISM ULTRIX and PRISM MICA operating systems.

This standard applies to all standard procedures. That is,

- All externally callable interfaces in DIGITAL-supported, standard system software
- All intermodule calls to major PRISM 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).

22.1 Properties That Must Be Held Invariant

Some conventions in this standard define properties that must be maintained at every point during execution of user mode code. These are properties which the underlying hardware architecture and/or software architecture depend on, and a program that violates any of these properties at any point may fail or may produce incorrect results.

- Frame Pointer (FP)

  If FP<2> is 1, then FP<31:3> must specify the address of an invocation descriptor for a procedure with a register frame. The contents of the registers specified by RA_SAVE, SP_SAVE, and FP_SAVE in that invocation descriptor must be valid as defined by this standard.

  If FP<2> is 0, then FP<31:3> must specify the address of a quadword aligned stack frame allocated on the thread stack, and all frame locations (including the saved register area) must be valid as defined by this standard.

- Stack Pointer (SP)

  SP must specify the quadword aligned address of the lowest valid address on the thread stack. All code must assume that, when the stack pointer is decremented, the contents of all addresses below the old (SP) are undefined until written; that is, there must be no "live" data on the stack at addresses lower than (SP).

  The difference between the value of SP and the address of some thread stack location known to be valid must never exceed GPLIM.

- Thread Environment Block Base (TEB_BASE)

  TEB_BASE must be zero (for a thread for which the TEB has not been initialized) or must specify the quadword aligned address of the thread environment block, and all TEB locations must be valid as defined by this standard.

- Invocation Descriptors

  Each invocation descriptor specified by the call stack or by FP must be quadword aligned, and each field in each descriptor must be valid as defined by this standard.

22.2 Properties Required of All Procedures

Some conventions in this standard define properties that are required for all procedure entry and return sequences. These are properties which the underlying software architecture depends on, and a procedure that violates any of these properties may cause the program to fail or produce incorrect results.

These requirements are in addition to those imposed on entry and return code sequences by the invariant properties described above.

Frame Activation

At the point that a procedure entry code sequence modifies the value of FP, all registers specified in the procedure's entry mask must have been saved in the called procedure's frame as defined by this standard.

Frame Deactivation

At the point that a procedure return code sequence modifies the value of FP, all registers specified in the procedure's entry mask must have been restored from the called procedure's frame as defined by this standard.

22.3 Properties Required for Standard Call and Return

Some conventions in this standard define properties that are required for all standard call and return interfaces. A *standard call* in this context is defined as any call that is not based on sufficient agreements between the calling and called procedure to permit these properties to be safely violated. Such standard calls include

- All externally callable interfaces in DIGITAL-supported, standard system software
- All intermodule calls to major PRISM 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).

These are properties which a calling or called user mode procedure may assume to be true, and a procedure that violates any of these properties may cause the program to fail or produce incorrect results.

These requirements are in addition to those imposed by the invariant properties and properties required of all procedures, described above.

Procedure Call

At the point that the JSR to a called procedure is executed, the following properties are required.

- The PRISM Call Conventions

  R10 (invocation descriptor address), R11 (return address), R13 (argument count), R14..R20 (argument list) and R12 (address of argument list) must have been set up as required by the conventions for a standard call.

- R4..R5

Registers R4..R5 must not contain data needed by either the calling or the called procedure, since they may be destroyed between the execution of the JSR and the execution of the first instruction of the called procedure.

- Argument List and Descriptor Structure

  The argument list (if any) must have been constructed as defined by this standard. All procedure values passed as arguments must produce the effect of an invocation descriptor when treated as an invocation descriptor by code that calls the procedure value. All descriptors for arguments or return values must conform to this standard.

- Memory Synchronization

  A DRAINM instruction must have been executed between any standard call and any unsafe vector load/store operation in the caller, both before and/or after the standard call.

Frame Activation

At the point that a procedure entry code sequence modifies the value of FP, the following properties are required.

- Preserved Registers

  Any scalar register R32..R63 which the called procedure may modify, or which any of the called procedure's descendents may modify without saving and restoring, must have been saved in the called procedure's frame as defined by this standard. The called procedure's entry mask must specify all of the saved registers and only the saved registers.

- Exception Synchronization

  If the called procedure has a condition handler, a DRAIN instruction must have been executed.

First Vector Load/Store

A DRAINM instruction must have been executed between the call to a procedure and the first unsafe vector load/store operation (if any) in the called procedure. (The DRAINM is typically executed in the entry code sequence, and must be executed before any vector registers are saved).

Frame Deactivation

At the point that a procedure return code sequence resets the value of FP, the following properties are required.

- Function Return Value

  The function return value (if any) must have been written as defined by this standard to R8; to R8..R9; to storage provided by the calling procedure; or to stack storage allocated by the calling procedure, and specified by a descriptor provided by the calling procedure and updated by the called procedure.

- Preserved Registers

  Any scalar register R32..R63 which the called procedure may modify, or which any of the called procedure's descendents may modify without saving and restoring, must have been restored from the called procedure's stack frame as defined by this standard.

- Memory Synchronization

A DRAINM instruction must have been executed between the last unsafe vector load/store operation (if any) in the called procedure and the return to the caller. (This DRAINM is typically executed in the return code sequence, and must be executed after any vector registers are restored).

- Exception Synchronization

  If the called procedure has a condition handler, a DRAIN instruction must have been executed after the last possible point of exception in the called procedure, but before resetting FP.

Procedure Return

At the point that the JSR to the calling procedure is executed, the following properties are required.

- Frame Pointer

  FP must have been reset to it's value on entry to the called procedure (that is, the called procedure's frame must have been deactivated).

- Stack Pointer

  SP must have been reset to it's value on entry to the called procedure, unless a function value is being returned on the stack, in which case SP must not have a higher value than it had on entry.

APPENDIX A

GUIDELINES FOR THE PRISM SOFTWARE ENVIRONMENT

This section describes guidelines for use of calling interfaces and mechanisms, modular programming recommendations, coding style guidelines, and other suggested practices.

This section of the document is *not* formally a part of the PRISM Extended Calling Standard. Rather, it is a separate standard with a scope and applicability similar to that of the VAX Modular Programming Standard.

Much of the information in this section is tenative, and comments from reviewers are encouraged and solicited.

A.1 Language Extensions for Argument Transmission

Since the PRISM calling standard permits arguments to be passed by immediate value, by reference, or by descriptor, language extensions are needed to reconcile the different argument passing mechanisms. In addition to the default passing mechanisms used, each language processor is required to give the user explicit control of the argument passing mechanism in the calling procedure for the data types supported by the language as follows.

Data Type	Immediate	Reference	Descriptor
Atomic $\leq$ 64 bits	Yes	Yes	No
Atomic $>$ 64 bits	No	Yes	No
Text String	No	Yes	Yes
Bit String	No	Yes	Yes
Array	No	Yes	Yes
Miscellaneous $\leq$ 64 bits	Yes	Yes	No
Miscellaneous $>$ 64 bits	No	Yes	No

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

%VAL(arg)	Immediate Mechanism—Corresponding argument item contains the 32-bit or 64-bit value of the argument, arg.
%REF(arg)	Reference Mechanism—Corresponding argument item contains the address of the value of the argument, arg.
%DESCR(arg)	Descriptor Mechanism—Corresponding argument item contains the address of the PRISM descriptor of the argument, arg.

These intrinsic functions can be used in the syntax of a procedure call to control generation of the argument list. For example:

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

A.2 Argument Data Types

The PRISM calling standard defines three classes of representational data types:

- Public interface data types
- Application-area data types
- Facility-specific data types

Each of the above classes is further organized into *atomic*, *string*, and *miscellaneous* data types.

Each language data type implemented by a high level language uses one of the these PRISM representational data types for procedure parameters and elements of file records. When existing data types are not sufficient to satisfy the semantics of a language, new data types will be added to this standard, including facility-specific data types.

Some data types are composed of a record-like structure consisting of two or more elementary data types. For example, the F_floating complex data type is made up of two F_floating data types, and the varying character coded text data type is made up of a word logical data type followed by a character coded text data type.

Unless explicitly 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.

\\
Exactly which data types should be in which class is a topic of continuing analysis. In particular, we want all the public interface types to work well with remote procedure calls. *These assignments are not final.*

Rigorous definition of the data types will be included in a future version of this standard. The public interface types and the application area types can be understood by reference to their VAX analogues and to other sections of this standard.
\\

A.2.1 Public Interface Data Types

Public interface data types are the preferred data types for use in all externally callable interfaces to DIGITAL-supported, standard system software.

Application-independent public interfaces should utilize only these data types.

All DIGITAL language processors must define the correspondence between language data types and these representational public interface types:

- Atomic Types

  boolean-valued byte
  signed 32-bit integer
  F_floating
  G_floating

- String Types

  Fixed-length character coded text

- Miscellaneous Types

  32-bit typed pointer

set of flags with ≤ 32 members
procedure value

```
\\
Note that procedure value may not work well for remote procedure calls.
This issue will be considered further.
\\
```

A.2.2 Application-area Data Types

Application-area data types are appropriate data types for use in externally callable interfaces, but are not recommended for application independent public interfaces.

These data types are common in certain applications and styles of programming, but are *not* fully supported by one or more important programming languages.

Therefore, although these data types are defined and supported for use in external interfaces and may be supported by multiple languages, they are considered specific to certain application areas.

DIGITAL language processors that support these representational data types must define the correspendence between language data types and these types:

- Atomic Types

  unsigned 32-bit integer
  signed 64-bit integer
  unsigned 64-bit integer
  F_floating complex
  G_floating complex

- String Types

  varying character coded text
  numeric string, left separate sign

- Miscellaneous Types

  32-bit untyped pointer (raw address)
  records that conform to PRISM record layout and alignment rules.
  records that conform to VAX record layout and alignment rules.

A.2.3 Facility-specific Data Types

Facility-specific data types are inappropriate for use in public interfaces, and are not recommended for interlanguage use.

They may be used freely for external procedure calls within a single language or facility, and when appropriate will be supported by language utilities such as the debugger.

However, these data types are not supported by most languages and/or do not work well for public interfaces and/or represent an unnecessary level of specification.

These representational data types include (but are not necessarily restricted to):

- Atomic Types

  signed 8-bit integer
  unsigned 8-bit integer

signed 16-bit integer
unsigned 16-bit integer
signed 128-bit integer
unsigned 128-bit integer
biased integer
intermediate temporary (COBOL)
D_floating
H_floating
D_floating complex
H_floating complex

D_floating and H_floating real and complex are further distinguished in that operations on these types are *not* directly supported by the PRISM hardware. D_floating and H_floating real should not be used.

- String Types

  dynamic strings
  two-byte character coded text
  varying two-byte character coded text
  aligned bit string
  unaligned bit string
  numeric string, unsigned
  numeric string, left overpunched sign
  numeric string, right separate sign
  numeric string, right overpunched sign
  numeric string, zoned sign
  packed decimal string
  picture (COBOL)
  ASCIC string (byte count)
  ASCIW string (word count)
  ASCIZ string (zero terminated)

- Miscellaneous Types

  parameterized types not passed by standard descriptor
  record that do not conform to either the PRISM or the VAX record layout and alignment rules
  address range
  bound label value
  self-relative label (PL/I)
  absolute date and time
  relative date and time
  condition value
  set with > 32 members
  typed pointer type spec
  untyped pointer type spec
  file
  record file address (BASIC)
  area (PL/I)
  block (BLISS)
  task (ADA)
  tree (SCAN)

tree pointer (SCAN)
unspecified

A.3 Other Guidelines for Arguments

Writeable storage passed for return of function values should be aligned according to the alignment rules of the return value's data type.

APPENDIX B

DESCRIPTOR DESIGN AND COMPATIBILITY NOTES

Argument descriptors are important for the following purposes:

1. Support of the semantics of high level languages (HLLs)
2. Support of existing VAX/VMS public interfaces for procedures that will be run on PRISM. These include: the language independent part of the RTL, VAX/VMS compatibility services, and call interfaces to utilities such as DTR and FMS.

The requirements of these 2 groups differ.

The first group poses requirements to support the semantics of various languages and to permit the passing of arguments between procedures written in different languages. For example, string descriptors are necessary to pass strings between procedures written in high level languages.

The requirements of the second group involve compatibility constraints. Public interfaces exist that users rely on. Removing information from descriptors potentially breaks these interfaces.

This standards addresses the needs of high level languages, most of the interfaces to the language independent part of the RTL, and VMS compatibility services.

It does *not* address the remaining interfaces outlined in group 2. Descriptors to support these interfaces will only be added if the need for them is substantiated.

The interfaces to the existing VMS language support RTLs are not considered in this standard since these interfaces are considered private to DIGITAL and thus can be changed.

The characteristics of VAX descriptors which have been eliminated in PRISM descriptors are:

1. Unimplemented combinations of CLASS and DTYPE

   The PRISM calling standard defines descriptors only for cases where they are needed. By merging CLASS and DTYPE, illegal combinations of these fields are eliminated.

2. Data types in array descriptors

   Since utility routines do not require array descriptors, array descriptors contain only the information needed to support high level language calls.

3. Descriptors for atomic types

   High level languages do not require descriptors to pass such objects. The VAX calling standard, however, has defined descriptors for these objects and most VAX languages can generate such atomic descriptors. Few languages (we know of none) will accept an atomic descriptor as a formal argument, and thus use of these descriptors is limited to communicating with utility routines.

It appears that the only public RTL interface that uses these descriptors is LIB$CVT_DXDX.

Therefore, descriptors for atomic types are not provided by this standard. They will be added only if the need for them is demonstrated.

4. Various data types

The following scalar data types cannot be described by PRISM descriptors:

Data Type	Reason
COBOL intermediate type	Private to COBOL
Instruction sequence type	Not needed to support HLLs
Entry mask type	Not needed to support HLLs
Bounded label value	Not needed to support HLLs
Bounded procedure value	Not needed to support HLLs
Absolute date-time	Not needed to support HLLs
Unspecified	Not needed to support HLLs
Descriptor	Private to certain languages
D and H floating point	Not supported by PRISM
Octaword	Not supported by PRISM
Aligned bit string	Merged with unaligned

5. Various descriptor classes

The following descriptor classes have been eliminated:

Class	Reason
Contiguous array descriptor	Merged with noncontiguous
Procedure descriptor	Not needed to support HLLs
Label descriptor	Not needed to support HLLs
Language specific descriptors	Private to certain languages
Decimal string descriptor	Not needed to support HLLs
Varying array string descriptor	Replaced by byte array descriptor
All deprecated VAX classes	No longer in use on the VAX

6. Various descriptor fields

The following descriptor fields have been eliminated:

Field	Reason
DTYPE	Not needed to support HLLs
SCALE	Not needed to support HLLs
DIGITS	Not needed to support HLLs
BINSCALE	Not needed to support HLLs
REDIM	Private to BASIC
ARSIZE	Can be derived from remaining fields

Field	Reason
POINTER	In byte arrays can be derived from remaining fields
COLUMN	Not needed for noncontiguous arrays

APPENDIX C

CALLING STANDARD TOPICS UNDER DEVELOPMENT

There are a number of calling standard conventions under development which have not yet been stabilized for inclusion in the main part of this document. These include:

- Function value return by optional dynamic string
- Thread local storage
- Thread local context
- Thread control and I/O synchronization
- Spin lock conventions
- Linkages to complex instruction sequences
- Rigorous definitions for data types

These conventions will be defined in future documents or revisions to this document. The current state of some of these topics is described in the following sections.

C.1 Function Value Return by Optional Dynamic String

The optional dynamic string mechanism supports returning the function value in dynamic string storage, but does *not require* use of dynamic string storage. The function value may, at the option of the called function, be returned by the top of stack mechanism.

The caller must pass as the first argument a function return descriptor which specifies the address of a valid dynamic string descriptor.

The fields of the function return descriptor must be initialized by the caller as follows.

CLASS = DESCR$C_DYNAMIC_RETURN
POINTER = the address of a valid dynamic string descriptor
EXTENT = *undefined*

The fields of the dynamic string descriptor must be initialized by the caller as follows.

CLASS = DESCR$C_DYNAMIC_TEXT
POINTER = the address of the first byte of storage
EXTENT = a signed integer $0..2^{31}-1$ specifying the length in bytes of the storage

If the caller does not provide any dynamic string storage for return of the function value, then the dynamic string descriptor must denote the null dynamic string.

Both descriptors must be allocated in writeable storage, and may be modified by the called function.

The called function must *either*

- proceed to step 1 of the top of stack mechanism, *or*

- assign the function return value to a dynamic string and update the dynamic string descriptor (not the function value descriptor) to denote the function return value as follows.

  CLASS = DESCR$C_DYNAMIC_TEXT
  POINTER = the address of the first byte of the return value
  EXTENT = a signed integer $0..2^{31} - 1$ specifying the length in bytes of the return value

No information is returned in registers R8..R9, and the function value descriptor is not modified.

If, and *only* if, the latter option is chosen, then all dynamic string management conventions apply to this operation. For example,

- All, part, or none of any dynamic string storage provided by the caller may be used by the called function to return its value.
- The called function may allocate new dynamic string storage in which to return its value.
- Any dynamic string storage provided by the caller which is not used to return the function value must be deallocated by the called function.
- All dynamic string allocation and deallocation by the called function must be done using the standard dynamic string management interfaces.

When control returns to the calling procedure, the caller must inspect the function return descriptor to determine which mechanism was used by the called function to return its value. The caller must manage the affected storage appropriately for the mechanism that was used.

For example, if the called function used the top of stack mechanism, then the caller must manage the return value according to that mechanism. In addition, since in this case the called function will have treated the function return descriptor as a STACK_RETURN descriptor, then the contents of the function return descriptor will have been destroyed by the called function and the descriptor will no longer specify the address of the dynamic string descriptor.

For this reason, calling procedures will normally maintain a copy of the address of the dynamic string descriptor passed to the called function, and use this copy to recover the dynamic string descriptor if the function returns its value by the top of stack machanism. In addition, since in this case the called function will not have utilized the dynamic string storage specified by the dynamic string descpritor, any storage specified by the dynamic string descriptor must be reclaimed by the caller if necessary.

C.2 Thread Local Static Storage

Thread local static storage (TLS) provides threads executing in a multithread environment with per-thread storage that has properties of process wide static storage (that is, the properties of static overlaid PSECTs):

- Any number of modules can contribute to a thread local static region; the length of the region is determined by linker processing of object code.
- A TLS region can initially contain zero or non-zero data. The initial contents of a TLS region are determined by linker processing of object code.
- A TLS region can be aligned as required by the contributing modules. The alignment of the region is determined by linker processing of object code.
- The offsets for data within a region are managed by compilers and the linker using the general methods available for managing data within PSECTs (such as variables within a COMMON block).

Thread local static storage has additional properties:

- It must be allocated and initialized when first referenced by a thread. Each thread referencing a TLS region references a distinct instance of the region.
- References to data within a TLS region are 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 references the region.
- Each TLS region allocated by a thread must be freed when the thread terminates.

All local static storage for a thread is managed via the TLS array, the base address of which is maintained in the thread's TEB.

The TLS array is an array of longwords. Each longword (except the first) specifies the location of a TLS region belonging to that thread. The first longword of the TLS array specifies the length of the array in bytes (for longwords both utilized and not yet utilized).

The offset of first free TLS descriptor, maintained the TEB, specifies the offset of the first byte in the first unused longword in the array.

Initially, the TLS array base address and first free offset are zero in the TEB.

Each distinct TLS region in the process has a unique offset in the TLS array. This offset is assigned by the linker and image activator in cooperation with one another: the image activator keeps track of the highest offset assigned so far; when a new image is activated the offset is incremented by 4 times the number of TLS regions in the image. The offsets are stored in a module's linkage section, and are referenced by special image fixup records.

The TLS array is not modified when an image is activated. Rather, whenever a thread needs to reference a TLS region, it determines whether that region exists by testing whether the first free TLS array offset specified by the TEB is greater than the offset assigned to that TLS region by the image activator, and whether the pointer in that array position has been initialized. (Normally this test will be done on entry to every procedure that references one or more TLS regions, and will be done once per local TLS region accessed by each procedure).

If either test fails, the thread must allocate and initialize the TLS region. That may be done, for example, by calling an RTL routine.

```
if R3^.TLS_ARRAY_FREE ≤  TLS_OFFSET
or else R3^.TLS_ARRAY_ADDRESS^[TLS_OFFSET] = 0
then
    CREATE_TLE_REGION(TLS_OFFSET,PROTOTYPE);

TLS_REGION_ADDRESS = TEB_BASE^.TLS_ARRAY_BASE^[TLS_OFFSET]
```

In this example, the RTL routine allocates memory for the specified TLS region, using the PROTOTYPE to determine the length, the alignment, and the initial contents (if any) of the region. If necessary, it also updates the first free TLS array offset in the TEB. If furthermore necessary, it allocates a new, larger TLS array, copies the old array into it, and updates the TLS array base in the TEB. It then initializes the new entry in the TLS array and returns.

(Note that the values of TLS_ARRAY_ADDRESS and TLS_ARRAY_FREE are not valid across external calls, since the array of TLS descriptors may be copied to a new location by an external procedure).

C.3 Thread Local Context

Thread local context is data and values that are unique to a thread.

This mechanism is provided for languages, applications, and utilities to manage language-specific thread control blocks, thread local virtual memory, lists of synchronization locks, etc.

Unlike thread local storage (TLS), which is designed and primarily intended for use by compilers to provide the correct language semantics in a shared-memory multithread environment, thread local context (TLC) is intended to provide per-thread context management to applications and high level language programmers via a modular run time library interface.

This mechanism supports a dynamic number of threads within a process, a dynamic amount of thread local context per thread, and local context in threads created by sharable images which were dynamically activated in a multithread environment. It provides thread local context management including allocation of resources, freeing of resources, coordination with other threads, and access by potentially all procedures executed by the thread.

A thread local context pointer contained by the TEB would be the root pointer to a data structure used to manage all thread local context not directly supported by this standard.

The proposed interface is a single routine with the following form:

longword = GET_THREAD_LOCAL_VALUE (IN key_block)

Key_block is an array of longwords as follows.

COUNT	REQUIRED: the number of longwords following
KEY	REQUIRED: a 32 bit value that uniquely identifies this local context within the thread; used by GET_THREAD_LOCAL_VALUE to distinguish this thread local context from any other within the thread
INIT_PROC	REQUIRED: an initialization procedure value to be called if GET_THREAD_LOCAL_VALUE has not previously seen this key within this thread
[TERM_PROC]	OPTIONAL: a termination procedure value to be called when the thread is terminated
[REF_PROC]	OPTIONAL: a reference procedure value to be called if GET_THREAD_LOCAL_VALUE has previously seen this key within this thread
[ARG1..ARGn]	OPTIONAL: 0..N longwords that are not directly interpreted by GET_THREAD_LOCAL_VALUE

When a thread requires the value of some thread local context, it calls GET_THREAD_LOCAL_VALUE, passing the address of a data structure called a key_block. The key_block describes the thread local context to be accessed. GET_THREAD_LOCAL_VALUE returns a longword which is the value for that thread local context.

GET_THREAD_LOCAL_VALUE maintains a per thread data structure of the keys that are currently defined for this thread. The root of this data structure is kept in the thread environment block.

If GET_THREAD_LOCAL_VALUE does not find the key in this database, it calls the initialization procedure specified in the key_block:

status = *INIT_PROC* (IN key_block, OUT longword)

The procedure specified by INIT_PROC performs any necessary initialization and returns the longword value that GET_THREAD_LOCAL_VALUE is to return to its caller.

Before returning, GET_THREAD_LOCAL_VALUE adds the new key to its data base. Associated with that key, it maintains the longword value from initialization procedure, and the termination procedure value (if any).

If GET_THREAD_LOCAL_VALUE does find the key in its database, it proceeds as follows.

If no reference procedure is specified, the longword maintained by GET_THREAD_LOCAL_VALUE for this key is returned.

If a reference procedure is specified, it is invoked to obtain a new value for GET_THREAD_LOCAL_VALUE to return.

status = *REF_PROC* (IN key_block, IN old_longword, OUT new_longword)

The reference procedure may examine the optional items ARG1..ARGn in the key_block. OLD_LONGWORD is the longword value associated with the key (OUT parameter of the INIT_PROC or last REF_PROC call). NEW_LONGWORD is the value for GET_THREAD_LOCAL_VALUE to return and to replace OLD_LONGWORD in the GET_THREAD_LOCAL_VALUE database.

The termination procedure maintained in the GET_THREAD_LOCAL_VALUE database are invoked when the thread terminates. If the TEB has a non-null pointer for the root of the GET_THREAD_LOCAL_VALUE database, a routine is called that will walk through the database and call the termination procedure (if specified) for each key.

status = *TERM_PROC* (IN longword)

If the key_block specified no termination procedure for a key, then no termination is performed for that key.

APPENDIX D

LITERAL VALUES FOR SYMBOLIC CONSTANTS