

# **PA-RISC Procedure Calling Conventions Reference Manual**

For the HP-UX and MPE XL Operating Systems

**HP 9000 Series 600/700/800 and  
HP 3000 Series 900 Computer Systems**



HP Part No. 09740-90015  
Printed in USA January 1991

Edition 2  
E0191

---

## **Notice**

**Hewlett-Packard makes no warranty of any kind with regard to this material, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose.** Hewlett-Packard shall not be liable for errors contained herein or for incidental or consequential damages in connection with the furnishing, performance, or use of this material.

Hewlett-Packard assumes no responsibility for the use or reliability of its software on equipment that is not furnished by Hewlett-Packard.

This document contains proprietary information which is protected by copyright. All rights are reserved. No part of this document may be photocopied, reproduced, or translated to another language without the prior written consent of Hewlett-Packard Company. The information contained in this document is subject to change without notice.

UNIX is a trademark of AT&T Laboratories in the USA and other countries.

**System Technology Division  
19483 Pruneridge Avenue  
Cupertino, CA 95014**

---

## Printing History

New editions are complete revisions of the manual. Update packages, which are issued between editions, contain additional and replacement pages to be merged into the manual by the customer. The dates on the title page change only when a new edition or a new update is published. No information is incorporated into a reprinting unless it appears as a prior update; the edition does not change when an update is incorporated.

The software code printed alongside the date indicates the version level of the software product at the time the manual or update was issued. Many product updates and fixes do not require manual changes and, conversely, manual corrections may be done without accompanying product changes. Therefore, do not expect a one-to-one correspondence between product updates and manual updates.

Edition 1 . . . . .	November 1986 . . . . .	E1186
Edition 2 . . . . .	January 1991 . . . . .	E0191

## List of Effective Pages

The List of Effective Pages gives the date of the current edition and of any pages changed in updates to that edition. Within the manual, any page changed since the last edition is indicated by printing the date the changes were made on the bottom of the page. No information is incorporated into a reprinting unless it appears as a prior update.

Effective Pages: All . . . . . January 1991

---

## Safety and Regulatory Information

For your protection this product has been tested to various national and international regulations and standards. The scope of this regulatory testing includes electrical/mechanical safety, radio frequency interference, ergonomics, acoustics, and hazardous materials. Where required, approvals obtained from third-party test agencies are shown on the product label. In addition, various regulatory bodies require some of the information under the following headings.

### USA Radio Frequency Interference

The United States Federal Communications Commission (in 47CFR Subpart J, of Part 15) has specified that the following notice be brought to the attention of the users of this product:

---

#### Warning



**This equipment generates, uses, and can radiate radio frequency energy and if not installed and used in accordance with the instructions manual, may cause interference to radio communications. It has been tested for compliance with the limits for Class A computing devices pursuant to Subpart J of Part 15 of FCC Rules, which are designed to provide reasonable protection against such interference. Operation of this equipment in a residential area is likely to cause interference in which case the user at his own expense will be required to take whatever measures may be required to correct the interference.**

---

### Japanese Radio Frequency Interference

The following notice is for users of this product in Japan:

この装置は、第一種情報装置(商工業地域において使用されるべき情報装置)で商工業地域での電波障害防止を目的とした情報処理装置等電波障害自主規制協議会(VCCI)基準に適合しております。

従って、住宅地域またはその隣接した地域で使用すると、ラジオ、テレビジョン受信機等に受信障害を与えることがあります。

取扱説明書に従って正しい取り扱いをして下さい。

#### Japanese Radio Frequency Notice

---

**Warning****UNITED KINGDOM TELECOM WARNING**

(United Kingdom Only)

Interconnection of ports marked “UNITED KINGDOM TELECOM WARNING: Connect only apparatus complying with BS 6301 to these ports” with ports not so marked may produce hazardous conditions on the network and advice should be obtained from a competent engineer before such a connection is made.

Connect only apparatus complying with BS 6301 to the ports marked with the above warning.

---

## Safety Considerations

This product and related documentation must be reviewed for familiarization with safety markings and instructions before operation. The following figure shows some of the safety symbols used on the product to indicate various safety considerations.

### SAFETY SYMBOLS



Instruction manual symbol: the product will be marked with this symbol when it is necessary for the user to refer to the instruction manual in order to protect the product against damage.



Indicates hazardous voltages.



Indicates earth (ground) terminal (sometimes used in manual to indicate circuit common connected to grounded chassis).

---

#### Warning



The **WARNING** sign denotes a hazard. It calls attention to a procedure, practice, or the like, which, if not done correctly or adhered to, could result in injury. Do not proceed beyond a **WARNING** sign until the indicated conditions are fully understood and met.

---

---

#### Caution



The **CAUTION** sign denotes a hazard. It calls attention to an operating procedure, practice, or the like, which, if not done correctly or adhered to, could damage or destroy part or all of the product. Do not proceed beyond a **CAUTION** sign until the indicated conditions are fully understood and met.

---





# Contents

---

<b>1. Overview</b>	
1.1. Types of Procedure Calls . . . . .	1-3
1.2. Interfaces . . . . .	1-4
<b>2. Stack Usage</b>	
2.1. Leaf and Non-Leaf Procedures . . . . .	2-1
2.2. Storage Areas Required for a Call . . . . .	2-2
2.3. Frame Marker Area . . . . .	2-4
2.4. Fixed Arguments Area . . . . .	2-5
2.5. Variable Arguments Area . . . . .	2-5
<b>3. Register Usage and Parameter Passing</b>	
3.1. Register Partitioning . . . . .	3-1
3.2. Other Register Conventions . . . . .	3-3
3.3. The Floating-Point Coprocessor Status Register . . . . .	3-4
3.4. Summary of Dedicated Register Usage . . . . .	3-4
3.5. Parameter Passing and Function Results . . . . .	3-6
3.5.1. Value Parameters . . . . .	3-6
3.5.2. Inter-Language Parameter Data Types and Sizes . . . . .	3-6
3.5.3. Reference Parameters . . . . .	3-7
3.5.4. Value-Result and Result Parameters . . . . .	3-7
3.5.5. Routine References . . . . .	3-8
3.5.6. Argument Register Usage Conventions . . . . .	3-8
3.5.7. Function Return Values . . . . .	3-9
3.5.8. Parameter Type Checking . . . . .	3-9
3.6. Parameter Relocation . . . . .	3-10
<b>4. The Actual Call</b>	
4.1. Control Flow of a Standard Procedure Call . . . . .	4-1
4.2. Efficiency . . . . .	4-2
4.3. The Code Involved in a Simple Local Call . . . . .	4-3
<b>5. Inter-Module Procedure Calls</b>	
5.1. External Calls on MPE XL . . . . .	5-1
5.1.1. Requirements of an External Call on MPE XL . . . . .	5-2
5.1.2. Requirements of an External Return . . . . .	5-2
5.1.3. Control Flow of an MPE XL External Call . . . . .	5-3
5.1.4. Calling Code . . . . .	5-3
5.1.5. Called Code . . . . .	5-4
5.1.6. Import and Export Stubs . . . . .	5-4
5.1.7. Import Stubs . . . . .	5-5
5.1.8. External Procedure Call Millicode (CALLX) . . . . .	5-6

5.1.9. Export Stubs . . . . .	5-6
5.1.10. Inter-Module Cross Reference Table (XRT) . . . . .	5-6
5.1.11. The Layout of the XRT . . . . .	5-7
5.1.12. Sub-table Header . . . . .	5-7
5.1.13. Linkage Pointer . . . . .	5-9
5.1.14. System Security and the XRT . . . . .	5-9
5.1.15. Interface Between Import and Export Stubs . . . . .	5-10
5.1.16. Summary of an External Procedure Call . . . . .	5-10
5.1.17. Dynamic Linking . . . . .	5-12
5.1.18. Procedure Labels . . . . .	5-12
5.2. External Calls on HP-UX . . . . .	5-13
5.2.1. Control Flow of an HP-UX External Call . . . . .	5-14
5.2.2. Calling Code . . . . .	5-15
5.2.3. Called Code . . . . .	5-16
5.3. PIC Requirements for Compilers and Assembly Code . . . . .	5-17
5.3.1. Long Calls . . . . .	5-17
5.3.2. Procedure Labels and Dynamic Calls . . . . .	5-18
<b>6. Millicode Calls</b>	
6.1. The Millicode Hierarchy . . . . .	6-2
6.2. Descriptions . . . . .	6-3
6.3. Introduction to Local and External Millicode . . . . .	6-3
6.4. Efficiency Factors . . . . .	6-4
6.5. Making a Millicode Call . . . . .	6-5
6.6. Nested Millicode Calls . . . . .	6-5
<b>7. Stack Unwinding</b>	
7.1. Requirements for Successful Stack Unwinding . . . . .	7-2
7.2. Unwinding From Millicode . . . . .	7-3
7.3. Instances in Which Unwinding May Fail . . . . .	7-3
7.4. Callee-Saves Register Spill . . . . .	7-4
7.5. The HP Pascal ESCAPE Mechanism . . . . .	7-4
7.6. A Simple Example . . . . .	7-5
7.7. An Advanced Example: Multiple Unwind Regions . . . . .	7-6
<b>A. Standard Procedure Calls</b>	
A.1. Pascal Source Code . . . . .	A-1
A.2. ANSI C Source Code . . . . .	A-2
A.3. FORTRAN Source Code . . . . .	A-3
A.4. Assembly Listing . . . . .	A-4
A.5. Code Description . . . . .	A-7
A.6. Other Compiler-Generated Information . . . . .	A-9
A.7. External Calls . . . . .	A-9
A.7.1. Pascal Source Code . . . . .	A-10
A.7.2. Assembly Code . . . . .	A-11
A.8. Assembly Documentation . . . . .	A-12

<b>B. Summary of PA-RISC Assembler Procedure Control</b>	
<b>C. The Stack Unwind Library</b>	
C.1. The Unwind Descriptor . . . . .	C-1
C.2. Unwind Utility Routines . . . . .	C-4
C.3. Recover Utility Routines . . . . .	C-7
C.3.1. The Recover Table . . . . .	C-7
C.4. Obtaining a Stack Trace . . . . .	C-8
C.5. Errors From The Unwind Library . . . . .	C-9
<b>D. The Stack Unwind Process</b>	
D.1. Role of Stubs in Unwinding . . . . .	D-1
D.2. Unwinding From Stubs on MPE XL . . . . .	D-2
D.3. Unwinding From Stubs on HP-UX . . . . .	D-2
D.4. Unwinding From Millicode . . . . .	D-3
D.5. Unwinding Across an Interrupt Marker . . . . .	D-3
D.6. Advanced Example . . . . .	D-4

## Figures

---

1-1. Procedure Call Convention Architected Interfaces. . . . .	1-4
2-1. General Stack Layout. . . . .	2-2
3-1. Register Partitioning. . . . .	3-2
3-2. Floating-Point Registers. . . . .	3-3
3-3. Parameter Relocation Stub. . . . .	3-11
4-1. Control Flow of a Standard Procedure Call. . . . .	4-2
5-1. Simplified External Procedure Call. . . . .	5-3
5-2. Layout of the XRT. . . . .	5-7
5-3. Sub-table Header. . . . .	5-7
5-4. One XRT Entry. . . . .	5-8
5-5. Summary of External Procedure Call. . . . .	5-11
5-6. HP-UX External Procedure Call. . . . .	5-15
6-1. Millicode Overview. . . . .	6-2
6-2. Millicode Stack Storage Layout. . . . .	6-6

## Tables

---

2-1. Elements of Single Stack Frame Necessary for a Procedure Call. . . . .	2-3
3-1. Space Register Usage. . . . .	3-4
3-2. General Register Usage. . . . .	3-5
3-3. Floating-Point Register Usage. . . . .	3-5
3-4. Parameter Data Types and Sizes. . . . .	3-6
3-5. Argument Register Use. . . . .	3-8
3-6. Return Values. . . . .	3-9

## Overview

---


Modern programming technique encourages programmers to practice well-structured decomposition, which entails the use of a greater number of smaller, more specialized procedures rather than larger, more complex routines. While this creates more adaptable and understandable programs, it also increases the frequency of procedure calls, thus making the efficiency of the procedure calling convention crucial to overall system performance.

Many modern machines provide instructions to perform many of the tasks necessary to make a procedure call, but this is not the case in Precision Architecture RISC (PA-RISC). Instead of using an architected mechanism, the procedure call is accomplished through a software convention which uses the machine's simple hardwired instructions, a solution that ultimately provides more flexibility and efficiency than the more complex (microcoded) instruction set additions.

Besides the obvious branch-and-return interruption that occurs in the flow of control as a result of a procedure call, many other provisions must be made in order to achieve an effective calling convention. The call mechanism must also pass parameters, save the caller's environment, and establish an environment for the called procedure (the *callee*). The procedure return mechanism must restore the caller's previous environment and save any return values.

Although PA-RISC machines are essentially register-based, by convention a stack is necessary for data storage. As a basis for discussion of the Procedure Calling Convention, we will first examine a straightforward calling mechanism in this environment, one in which the calling procedure (caller) acquires the responsibility for preserving its own state. This simplified model employs the following steps for each call:

---

**Note**  These steps are *NOT* the exact implementation used in the PA-RISC, but are given as a general basis for the discussion of the actual Procedure Calling Convention that will follow.

---

- Save all registers whose contents must be preserved across the procedure call. This prevents the callee, which will also use and modify registers, from affecting the caller's state. On return, those register values are restored.
- Evaluate parameters in order and push them onto the stack. This makes them available to the callee, which, by convention, knows how to access them.
- Push a frame marker, which is a fixed-size area containing several pieces of information. Included is the static link, which provides information needed by the callee in order to address the local variables and parameters of the caller, as well as the return address of the caller.
- Branch to the entry point of the callee.

And to return from a call in this model, it is necessary that:

- The callee extract the return address from the frame marker and branch to it, and
- The caller then remove the parameters from the stack and restore all saved registers before the program flow continues.

This model correctly implements the basic steps needed to execute a procedure call, but is relatively expensive. The caller is forced to assume all responsibility for preserving its state, which is a conservative and safe approach, but causes an excessive number of register saves to occur. To optimize the program's execution, the compiler makes extensive use of registers to hold local variables and temporary values; these registers must all be saved at a procedure call and restored at the return. A high overhead is also incurred by the loading and storing of parameters and linkage information. The procedure call convention implemented in PA-RISC focuses on the need to reduce this expense by maximizing register usage and minimizing direct memory references.

PA-RISC compilers attempt to alleviate this problem by introducing a procedure call mechanism that divides the register sets into *partitions*. The registers are partitioned into *caller-saves* (the caller is responsible for saving and restoring them), *callee-saves* (the callee must save them at entry and restore them at exit), and *linkage* registers. In the general purpose register set, sixteen of the registers comprise the callee-saves partition and thirteen are available for use as caller-saves registers.

Thus the responsibility for saving registers is divided between the caller and the callee, and some registers are also available for linkage. The floating-point registers and space registers are also partitioned in a similar manner.

The register allocator avoids unnecessary register saves by using caller-saves registers for values that need not be preserved across a call, while values that must be preserved are placed into registers from the callee-saves partition. At procedure entry, only those callee-saves registers used in the procedure are saved; this minimizes the number and frequency of register loads and stores during the course of a call. If more registers are needed from a particular partition than are available, registers can be borrowed from the other partition. The penalty for using these additional registers is that they must be saved and restored, but this overhead is incurred only in the special circumstance where excess registers are needed, which happens relatively infrequently.

In the simple model outlined above, all parameters are passed by being placed on the stack, which is expensive because direct memory references are needed in order to push each parameter. In PA-RISC procedure calling convention, this problem is lessened by the compilers, which allocate a permanent parameter area (in memory) large enough to hold the parameters for all calls performed by the procedure, and minimize memory references when storing parameters by using a combination of registers and memory to pass parameters. Four registers from the caller-saves partition are used to pass user parameters, each holding a single 32-bit value or half of a 64-bit value. Since procedures frequently have few parameters, the four registers are usually enough to accommodate them all. This removes the necessity of storing parameter values in the parameter area before the call. If more than four 32-bit parameters are passed, the additional ones are stored in the preallocated parameter area, or if a parameter is larger than 64 bits, its address is passed and the callee copies it to a temporary area.

Additional savings on memory access are gained when the callee is a leaf procedure (one that does not make any other calls). In this situation, the register allocator uses the caller-saves registers to hold variable values, thus eliminating the need to save callee-saves registers that it

might have used in a non-leaf procedure. Furthermore, since a leaf procedure will not make subsequent procedure calls, there is no need to allocate a stack frame for it, because the return address and other values can remain in registers during the entire life of the call. (Actually, there are rare exceptions to this; a stack frame may be necessary for a leaf procedure if more local space is needed than is available in registers.)

---

## 1.1. Types of Procedure Calls

Procedure calls can be grouped into three categories, depending on the location of the callee. The possibilities are:

1. Procedures residing in the same load module (intra-module call)
2. Procedures residing in other load modules (inter-module call)
3. Operating system or subsystem procedures

---

### Note

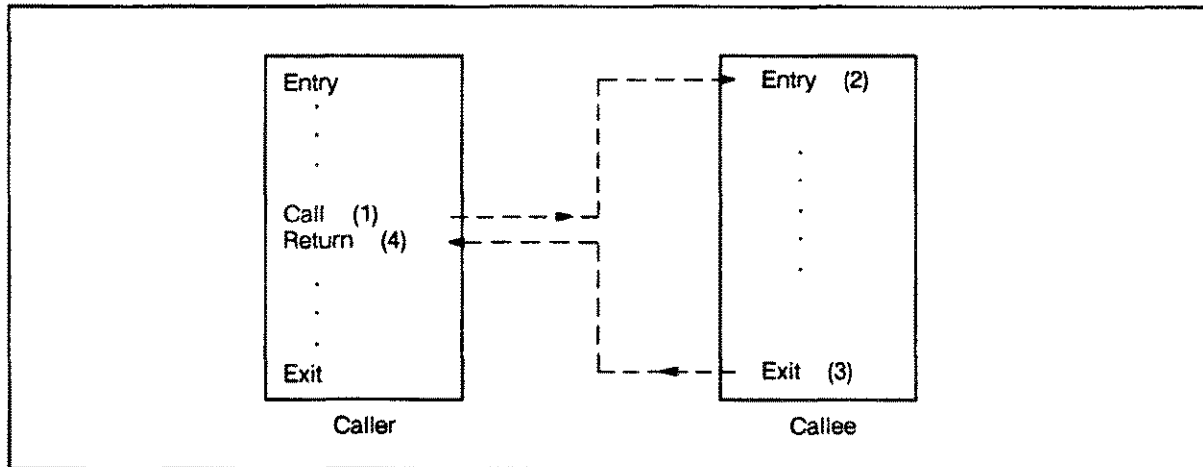


Throughout the rest of this document, *local* is used as a synonym for *intra-module* and *external* is used interchangeably with *inter-module* or *OS/Subsystem*.

---

In order to simplify code generation, all three types of calls are mapped into the local (intra-module) case, thus requiring the compiler to recognize only a single type of call. This is accomplished through the use of two types of *stubs*—*calling stubs* and *called stubs*—which establish the external branching and linking necessary for inter-module calls. Inter-module calls and specific stub usages are discussed in more detail in Chapter 5, Inter-Module Procedure Calls.

## 1.2. Interfaces



LG200010\_011

**Figure 1-1. Procedure Call Convention Architected Interfaces.**

Figure 1-1 shows the units involved in a procedure call, and the critical interfaces (numbered) that will be discussed throughout this document, and their definitions follow:

- Caller: The calling code, origin of the call.
- Callee: The called code, object of the call.
- Call: The transfer of program control to the callee (1).
- Entry: The point of entry into the callee (2).
- Exit: The point of exit from the callee (3).
- Return: The point of return of program control to the caller (4).



## Stack Usage

---

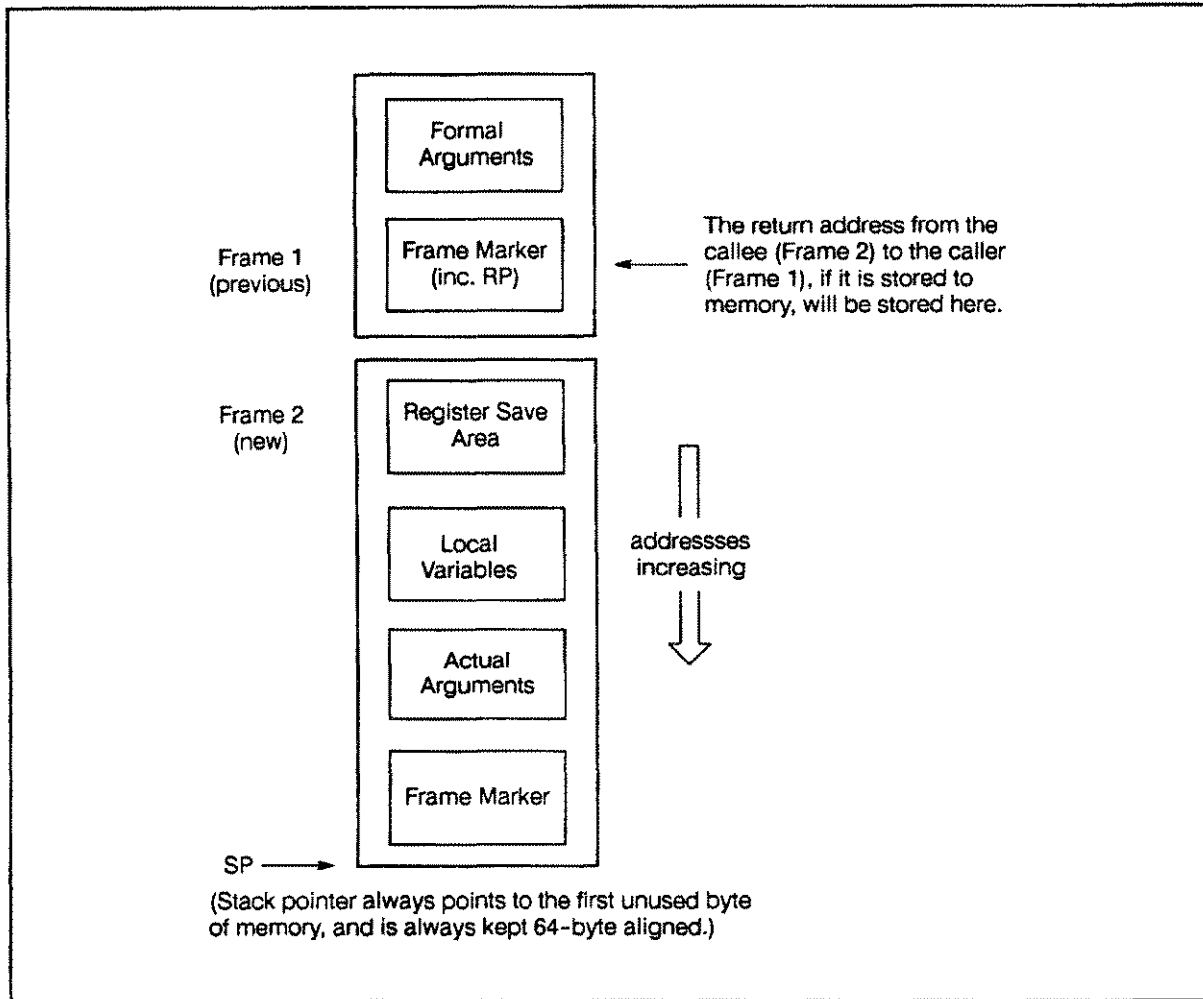
Because no explicit procedure call stack exists in the PA-RISC processor architecture, the stack is defined and manipulated entirely by software convention. When a process is initiated by the operating system, a virtual address range is allocated to that process to be used for the call stack. The stack pointer (gr30) is initialized to point to the low end of this range. As procedures are called, the stack pointer is incremented to allow the called procedure's frame to exist at the addresses below the stack pointer. As procedures are exited, the stack pointer is decremented by the same amount. The stack pointer always points to the first word above the current frame.

---

### 2.1. Leaf and Non-Leaf Procedures

All procedures can be classified in one of two categories: leaf or non-leaf. A leaf procedure is one that makes no additional calls, while a non-leaf procedure is one that does make additional calls. Although simple, the distinction is essential because the two cases entail considerably different requirements regarding (among other things) stack allocation and usage. Every non-leaf procedure requires the allocation of an additional stack frame in order to preserve necessary execution values and arguments. A stack frame is not always necessary for a leaf procedure. The recognition of a procedure as fitting into either the leaf or non-leaf category and the determination of the necessary frame size is done at compile time.

As will become evident throughout this document, it is often the case that much of a procedure's state information is saved in the caller's frame. This helps to avoid unnecessary stack usage. A general picture of the top of the stack for one call, including the frames belonging to the caller (previous) and callee (new), is shown in Figure 2-1.



LG200010\_012

**Figure 2-1. General Stack Layout.**

## 2.2. Storage Areas Required for a Call

The elements of a single stack frame that must be present in order for a procedure call to occur are shown below. The stack addresses are all given as byte offsets from the actual SP (stack pointer) value; for example, 'SP-36' designates the address 36 bytes below the current SP value.

The fields shown in Table 2-1 are explained in the text following the table.

**Table 2-1. Elements of Single Stack Frame Necessary for a Procedure Call.**

Offset	Contents	
Variable Arguments (optional; any number may be allocated)		
SP-(4*(N+9))	arg word N	
:	:	
:	:	
SP-56	arg word 5	
SP-52	arg word 4	
Fixed Arguments (must be allocated; may remain unused)		
SP-48	arg word 3	
SP-44	arg word 2	
SP-40	arg word 1	
SP-36	arg word 0	
Frame Marker		
SP-32	External Data/LT Pointer (LPT)	(set before Call)
SP-28	External sr4/LT Pointer (LPT')	(set after Call)
SP-24	External/stub RP (RP')	(set after Call)
SP-20	Current RP	(set after Entry)
SP-16	Static Link	(set before Call)
SP-12	Clean Up	(set before Call)
SP- 8	Relocation Stub RP (RP'')	(set after Call)
SP- 4	Previous SP	(set before Call)
Top of Frame		
SP- 0	Stack Pointer (points to next available address)	
	< top of frame >	

The size of a stack frame is required to be a multiple of 64 bytes so that the stack pointer is always kept 64-byte aligned. Since cache-lines on PA-RISC can be no larger than 64 bytes, this requirement allows compilers to know when data structures allocated on the stack are cache-line aligned. Knowledge of this alignment allows the compiler to use cache hints on memory references to those structures.

---

**Note**

Previous versions of this calling convention only required that stack frames be multiples of 8 bytes in size. The requirement has been increased to allow more effective use of cache hints on memory references to data items on the stack. Through the use of fixups specified by the compilers, the linker will be able to detect and eliminate uses of the cache hint bits in situations where 64-byte stack pointer alignment cannot be guaranteed. Some of the examples in this manual do not have 64-byte aligned stack frames.

---

## 2.3. Frame Marker Area

This eight-word area is allocated by any non-leaf routine prior to a call. The exact size of this area is defined because the caller uses it to locate the formal arguments from the previous frame. (Any standard procedure can identify the bottom of its own frame, and can therefore identify the formal arguments in the previous frame, because they will always reside in the region beginning with the ninth word below the top of the previous frame.)

*Previous SP:* Contains the old (procedure entry) value of the Stack Pointer. It is only required that this word be set if the current frame is noncontiguous with the previous frame, has a variable size, or is used with the static-link.

*Relocation Stub RP (RP'')*: Reserved for use by a relocation stub that must store a Return Pointer (RP) value, so the stub can be executed after the exit from the callee, but before return to the caller. See Chapter 3 for detailed discussion of Parameter Relocation stubs.

*Clean Up:* Area reserved for use by language processors; possibly for a pointer to any extra information (i.e. on the heap) that may otherwise be lost in the event of an abnormal interrupt.

*Static Link:* Used to communicate static scoping information to the callee that is necessary for data access. It may also be used in conjunction with the SL register, or to pass a display pointer rather than a static link, or it may remain unused.

*Current RP:* Reserved for use by the called procedure; this is where the current return address must be stored if the procedure uses RP (gr2) for any other purpose.

*External/Stub RP (RP'), External sr4/LTP', and External DP/LTP:* All three of these words are reserved for use by the inter-modular (external) calling mechanism. See Chapter 5 for more details.

---

## **2.4. Fixed Arguments Area**

These four words are reserved for holding the argument registers, should the callee wish to store them back to memory so that they will be contiguous with the memory-based parameters. All four words must be allocated for a non-leaf routine, but may remain unused.

---

## **2.5. Variable Arguments Area**

These words are reserved to hold any arguments that can not be contained in the four argument registers. Although only a few words are shown in this area in the diagram, there may actually be an unlimited number of arguments stored on the stack, continuing downward in succession (with addresses that correspond to the expression given in the diagram). Any necessary allocation in this area must be made by the caller.



## Register Usage and Parameter Passing

---

The PA-RISC processor architecture does not have instructions which specify how registers should be used or how parameter lists should be built for procedure calls. Instead, the software procedure calling convention prescribes the register usage and parameter passing guidelines.

---

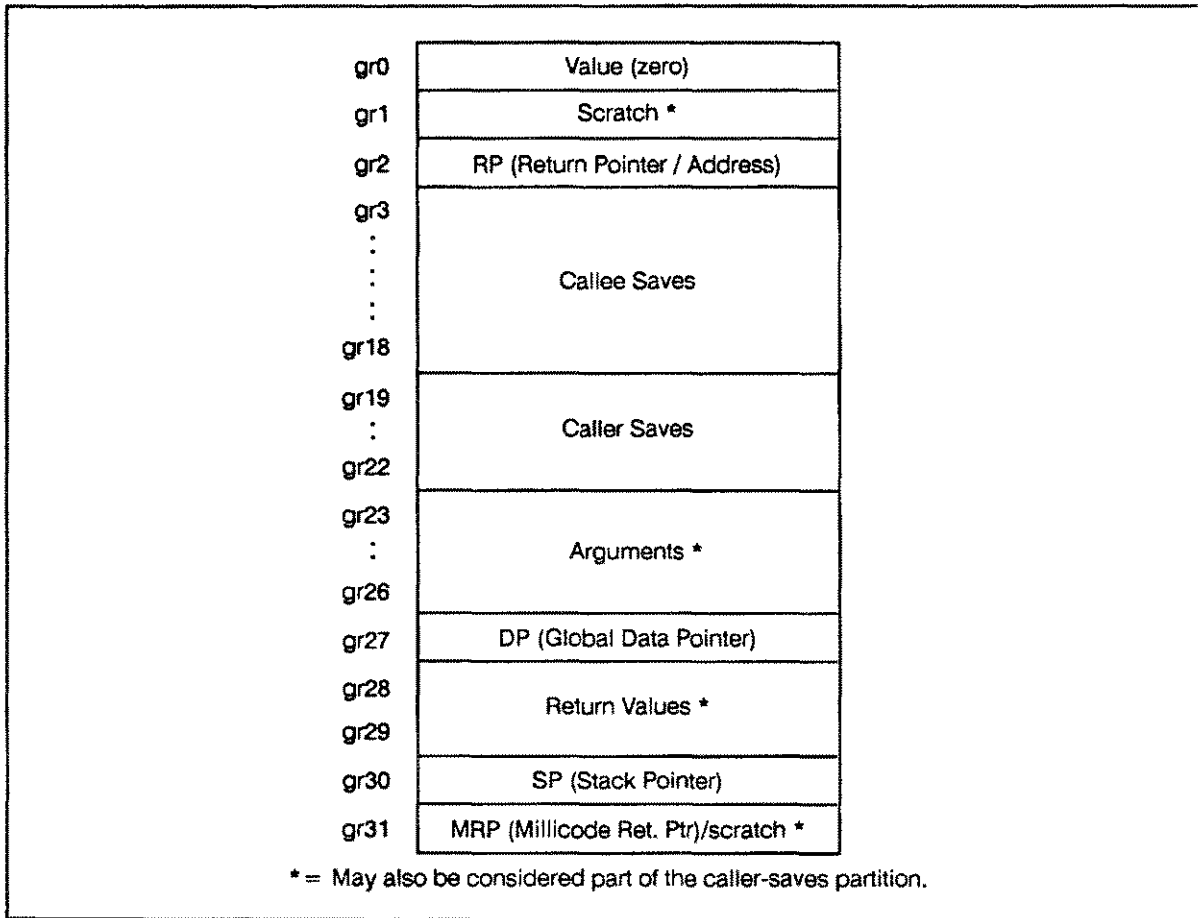
### 3.1. Register Partitioning

In order to reduce the number of register saves required for typical procedure calls, the PA-RISC general and floating-point register files have been divided into partitions designated as callee-saves and caller-saves. The names of these partitions indicate which procedure takes responsibility for preserving the contents of the register when a call is made.

If a procedure uses a register in the callee-saves partition, it must save the contents of that register immediately after procedure entry and restore the contents before the exit. Thus, the contents of all callee-saves registers are guaranteed to be preserved across procedure calls.

A procedure is free to use the caller-saves registers without saving their contents on entry. However, the contents of the caller-saves registers are not guaranteed to be preserved across calls. If a procedure has placed a needed value in a caller-saves register, it must be stored to memory or copied to a callee-saves register before making a call.

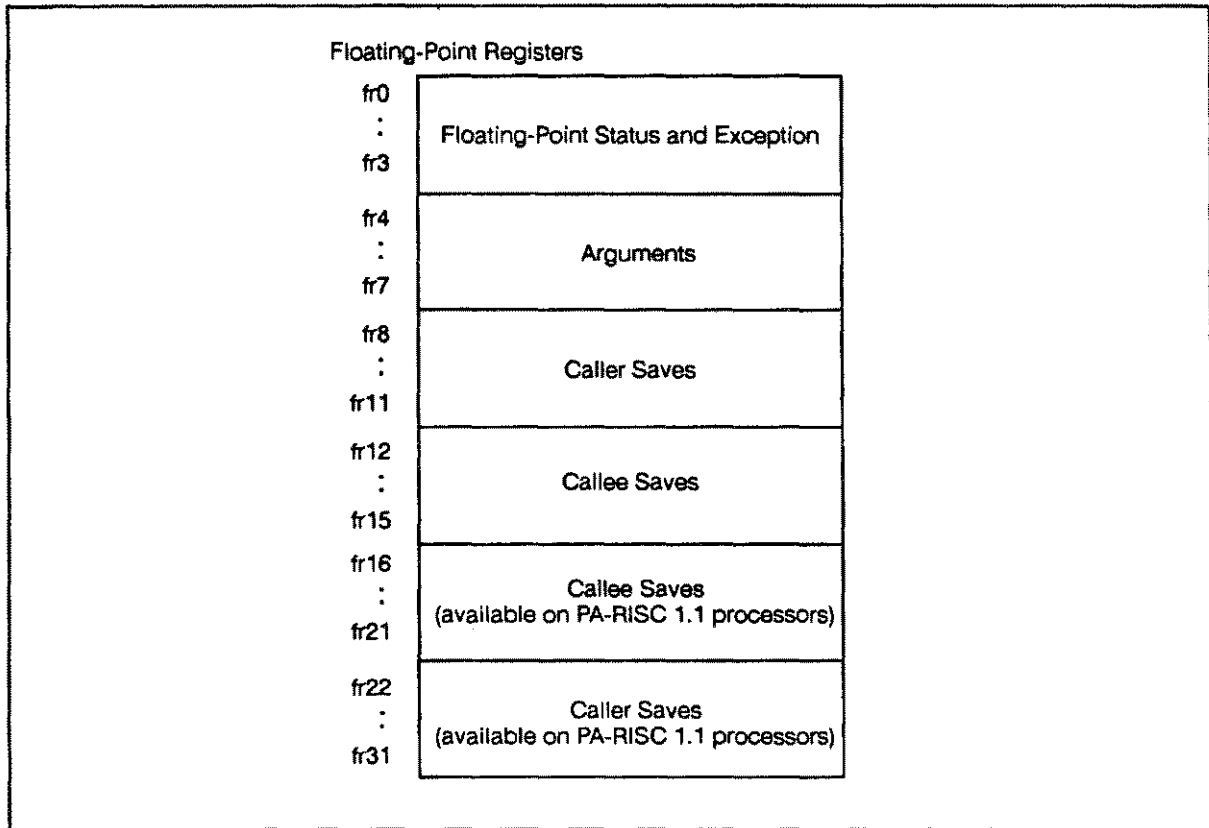
These partitions and the other dedicated registers are shown in Figures 3-1 and 3-2.



LG200010\_013

**Figure 3-1. Register Partitioning.**





**Figure 3-2. Floating-Point Registers.**

### 3.2. Other Register Conventions

The following are guaranteed to be preserved across calls:

- The procedure entry value of SP.
- The value of DP.
- Space registers sr3, sr4, sr5, sr6, and sr7.

The following are not necessarily preserved across calls:

- Floating-point registers fr1, fr2, and fr3.
- Space registers sr0, sr1, and sr2.
- The Processor Status Word (PSW).
- The shift amount register (cr11) or any control registers that are modified by privileged software (e.g. Protection IDs).
- The state, including internal registers, of any special function units accessed by the architected SPOP operations.

---

### 3.3. The Floating-Point Coprocessor Status Register

Within the floating-point coprocessor status register (fr0), the state of the rounding mode (bits 21-22) and exception trap enable bits (bits 27-31) are guaranteed to be preserved across calls. An exception to this convention is made for any routine which is defined to explicitly modify the state of the rounding mode or the trap enable bits on behalf of the caller.

The states of the compare bit (bit 5), the delayed trap bit (bit 25), and the exception trap flags (bits 0-4) are not guaranteed to be preserved across calls.

---

### 3.4. Summary of Dedicated Register Usage

Table 3-1, Table 3-2, and Table 3-3 show the required conventions regarding argument and return value passing. Users risk the unintentional destruction of necessary data if their programs do not adhere to the standard conventions.

**Table 3-1. Space Register Usage.**

Register Name	Other Names	Usage Convention
sr0		Caller-saves space register or millicode (nested or external) millicode return space register.
sr1	sarg sret	Space argument and return register or caller-saves space register.
sr2		Caller-saves space register.
sr3		Callee-saves space register.
sr4		Code space register (stubs save and restore on inter-module calls).
sr5		Data space register, modified only by privileged code.
sr6		System space register, modified only by privileged code.
sr7		System space register, modified only by privileged code.

---

#### Note



If the routine in question is a non-leaf routine, gr2 must be stored because subsequent calls will modify it. Once stored, it is available to be used as a scratch register by the code generators.

Although common, it is not absolutely necessary that gr2 be restored before exit; a branch (BV) using another caller-saves register is allowed.

**Table 3-2. General Register Usage.**

Register Name	Other Names	Usage Convention
gr0		Zero value register. (Writing to this register does not affect its contents.)
gr1		Scratch register (caller-saves). (can be destroyed by call mechanism).
gr2	RP	Return pointer and scratch register.
gr3 - gr18		General purpose callee-saves registers.
gr19 - gr22		General purpose caller-saves registers.
gr23	arg3	Argument register 3 or general purpose caller-saves register.
gr24	arg2	Argument register 2 or general purpose caller-saves register.
gr25	arg1	Argument register 1 or general purpose caller-saves register.
gr26	arg0	Argument register 0 or general purpose caller-saves register.
gr27	DP	Global data pointer; may not be used to hold other values. (Stubs save and restore on inter-module calls)
gr28	ret0	Function return register on exit or function result address on entry. May also be used as a general purpose caller-saves register.
gr29	SL ret1	Static link register (on entry), millicode function return or function return register for upper part of a 33 to 64 bit function result. May also be used as a general purpose caller-saves register.
gr30	SP	Stack pointer, may not be used to hold other values.
gr31		Millicode return pointer or scratch register (caller-saves).

**Table 3-3. Floating-Point Register Usage.**

Register Name	Other Names	Usage Convention
fr0		Floating-point coprocessor status. See discussion in the text of this chapter.
fr1 - fr3		Floating-point exception registers. Cannot be modified by user code.
fr4	fret farg0	Floating-point return register, single-precision argument register 0, or floating-point caller-saves register.
fr5	farg1	Single-precision argument register 1, double-precision argument register 0 or floating-point caller-saves register.
fr6	farg2	Single-precision argument register 2, or floating-point caller-saves register.
fr7	farg3	Single-precision argument register 3, double-precision argument register 1 or floating-point caller-saves register.
fr8 - fr11		Floating-point caller-saves registers.
fr12 - fr15		Floating-point callee-saves registers.
fr16 - fr21		Floating-point callee-saves registers, only available on PA-RISC version 1.1 or later processors.
fr22 - fr31		Floating-point caller-saves registers, only available on PA-RISC version 1.1 or later processors.

---

## 3.5 Parameter Passing and Function Results

### 3.5.1. Value Parameters

Value parameters are mapped to a sequential list of argument words with successive parameters mapping to successive argument words, except 64-bit parameters, which must be aligned on 64-bit boundaries. Irregularly sized data items should be extended to 32 or 64 bits. (The practice that has been adopted is to right-justify the value itself, and then left-extend it.) Non-standard length parameters that are signed integers are sign-extended to the left to 32 or 64 bits. This convention does not specify how 1-31, 33-63-bit data items are passed by value (except single ASCII characters).

Table 3-2 lists the sizes for recognized inter-language parameter data types. The form column indicates which of the forms (space ID, nonfloating-point, floating-point, or any) the data type is considered to be.

### 3.5.2. Inter-Language Parameter Data Types and Sizes

Table 3-4. Parameter Data Types and Sizes.

Type	Size (bits)	Form
ASCII character (in low order 8 bits)	32	Nonfloating-Pt.
Integer	32	Nonfloating-Pt. or Space ID
Short Pointer	32	Nonfloating-Pt.
Long Pointer	64	Nonfloating-Pt.
Routine Reference (see below for details of Routine Reference)	32 or 64	Routine Reference
Long Integer	64	Nonfloating-Pt.
Real (single-precision)	32	Floating-Pt.
Long Real (double-precision)	64	Floating-Pt.
Quad Precision	128	Any

Space Identifier (SID) (32 Bits): One arg word, callee cannot assume a valid SID.

Non-Floating-Point (32 Bits): One arg word.

Non-Floating-Point (64 Bits): Two words, double word aligned, high order word in an odd arg word. This may create a void in the argument list (i.e. an unused register and/or an unused word on the stack.)

Floating-Point (32 Bits, single-precision): One word, callee cannot assume a valid floating-point number.

Floating-Point (64 Bits, double-precision): Two words, double word aligned (high order word in odd arg word). This may create a void in the argument list. 64-bit floating-point value parameters mapped to the first and second double-words of the argument list should be passed in `farg1` and `farg3`, respectively. `farg0` and `farg2` are never used for 64-bit floating-point parameters. Callee cannot assume a valid floating-point number.

---

**Note**

The point is made that the callee “cannot assume a valid” value in these cases because no specifications are made in this convention that would ensure such validity.

---

Any Larger Than 64 Bits: A short pointer (using `sr5 - sr7`) to the high-order byte of the value is passed as a nonfloating-point 32-bit value parameter. The callee must copy the accessed portion of the value parameter into a temporary area before any modification can be made to the (caller's) data. The callee may assume that this address will be aligned to the natural boundary for a data item of the parameter's type. It should be noted that some compilers support options which allow data structures to be aligned on non-natural boundaries. The instruction sequence used to copy the value should be consistent with the data alignment assumptions made by potential callers of that routine.

---

**Note**

The natural boundaries for data types on PA-RISC are documented in the Programmer's Guide that is available for each supported programming language.

---

### 3.5.3. Reference Parameters

A short pointer to the referenced data item (using `sr4 - sr7`) is passed as a nonfloating-point 32-bit value parameter. The alignment requirements for the short pointer are the same as those mentioned for value parameters larger than 64 bits. Note that `sr4` can only be used if the call is known to be local, because an external call will modify `sr4`. (See Chapter 5 for further details.)

### 3.5.4. Value-Result and Result Parameters

It is intended that language processors can use either the reference or value parameter mechanisms for value-result and result parameters. In particular, Ada uses the argument registers/parameters as output registers/parameters.

### 3.5.5. Routine References

This convention requires that routine references (i.e. procedure parameters, function pointers, external subroutines) be passed as 32-bit nonfloating-point values.

It is expected that language processors that require a static link to be passed with a routine reference (i.e. Pascal passing level 2 procedures) will pass that static link as a separate 32-bit nonfloating-point value parameter. A language processor is free to maximize the efficiency of static scope linking within the requirements, without impacting other language processors. (Pascal passes routine references as either two separate 32-bit values or as one 64-bit value.) See Chapter 5 for further details on Routine References.

### 3.5.6. Argument Register Usage Conventions

Parameters to routines are logically located in the argument list. When a call is made, the first four words of the argument list are passed in registers, depending on the usage and number of the argument. The first four words of the actual argument list on the stack are reserved as spill locations for the argument registers. These requirements imply that the minimum argument list size is 16 bytes; this space must be allocated in the frame for non-leaf procedures, but it may remain unused.

The standard argument register use conventions are shown in Table 3-3.

**Table 3-5. Argument Register Use.**

	void	SID	nonFP	FP32	FP64
arg word 0	no reg	sarg	arg0	farg0	farg1 {32..63}
arg word 1	no reg	arg1	arg1	farg1	farg1 {0..31}
arg word 2	no reg	arg2	arg2	farg2	farg3 {32..63}
arg word 3	no reg	arg3	arg3	farg3	farg3 {0..31}

definitions:

- void - arg word not used in this call
- SID - space identifier value
- nonFP - any 32-bit or 64-bit nonfloating-point
- FP32 - 32-bit floating-point (single-precision)
- FP64 - 64-bit floating-point (double-precision)

### 3.5.7. Function Return Values

Function result values are placed in registers as described in Table 3-4. As with value parameters, irregularly sized function results should be extended to 32 or 64 bits. (The practice that has been adopted is to right-justify the value itself, and then left-extend it.) Non-standard length function results that are signed integers are sign-extended to the left to 32 or 64 bits. This convention does not specify how 1 - 31 or 33 - 63-bit data items are returned (except single ASCII characters).

When calling functions that return results larger than 64 bits, the caller passes a short pointer (using sr5 - sr7) in gr28 (ret0) which describes the memory location for the function result. The address given should be the address for the high-order byte of the result. The function may assume that the result address will be aligned to the natural boundary for a data item of the result's type. It should be noted that some compilers support options which allow data structures to be aligned on non-natural boundaries. The instruction sequence used to store a function result should be consistent with the data alignment assumptions made by potential callers of that function.

**Table 3-6. Return Values.**

Type of Return Value	Return Register
ASCII character	ret0 (gr28) - low order 8 bits
Nonfloating-Pt. (32-bit)	ret0 (gr28)
Nonfloating-Pt. (64-bit)	ret0 (gr28) - high order word ret1 (gr29) - low order word
Floating-Pt. (32-bit)	fret (fr4) *
Floating-Pt. (64-bit)	fret (fr4) *
Space Identifier (32-bit)	sret (sr1)
Any Larger Than 64-bit	result is stored to memory at location described by a short pointer passed by caller in gr28**

\* Although not common, it is possible to return floating-point values in general registers, as long as the argument relocation bits in the symbol record are set correctly. (Refer to Parameter Relocation for more details.)

\*\* The caller may not assume that the result's address is still in gr28 on return from the function.

### 3.5.8. Parameter Type Checking

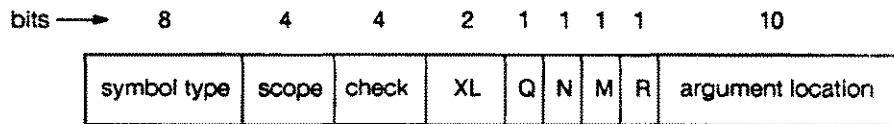
Some compilers may place argument descriptors in the object file which contain information about the type of each parameter passed and each formal argument expected. These descriptors are then checked by the linker for compatibility. If they do not match, a warning is generated. There is currently no mechanism available in the PA-RISC assembler to generate these argument descriptors.

### 3.6 Parameter Relocation

The procedure calling convention specifies that the first four words of the argument list and the function return value will be passed in registers: floating-point registers for floating-point values, general registers otherwise. However, some programming languages do not require type checking of parameters, which can lead to situations where the caller and the callee do not agree on the location of the parameters. Problems such as this occur frequently in the C language where, for example, formal and actual parameter types may be unmatched, due to the fact that no type checking occurs.

A parameter relocation mechanism alleviates this problem. The solution involves a short code sequence, called a relocation stub, which is inserted between the caller and the callee. When executed, the relocation stub moves any incorrectly located parameters to their expected location. If a procedure is called with more than one calling sequence, a relocation stub is needed for each non-matching calling sequence.

The compiler or assembler must communicate the location of the first four words of the parameter list and the location of the function return value to the linker and loader. To accomplish this, ten bits of argument location information have been added to the definitions of a symbol and a fix-up request. The following diagram shows the first word of a symbol definition record in the object file.



LG200010\_024

The argument location information is further broken down into five location values, corresponding to the first four argument words and the function return value, as shown below:

- Bits 22-23 : define the location of parameter list word 0
- Bits 24-25 : define the location of parameter list word 1
- Bits 26-27 : define the location of parameter list word 2
- Bits 28-29 : define the location of parameter list word 3
- Bits 30-31 : define the location of the function value return

The value of an argument location is interpreted as follows:

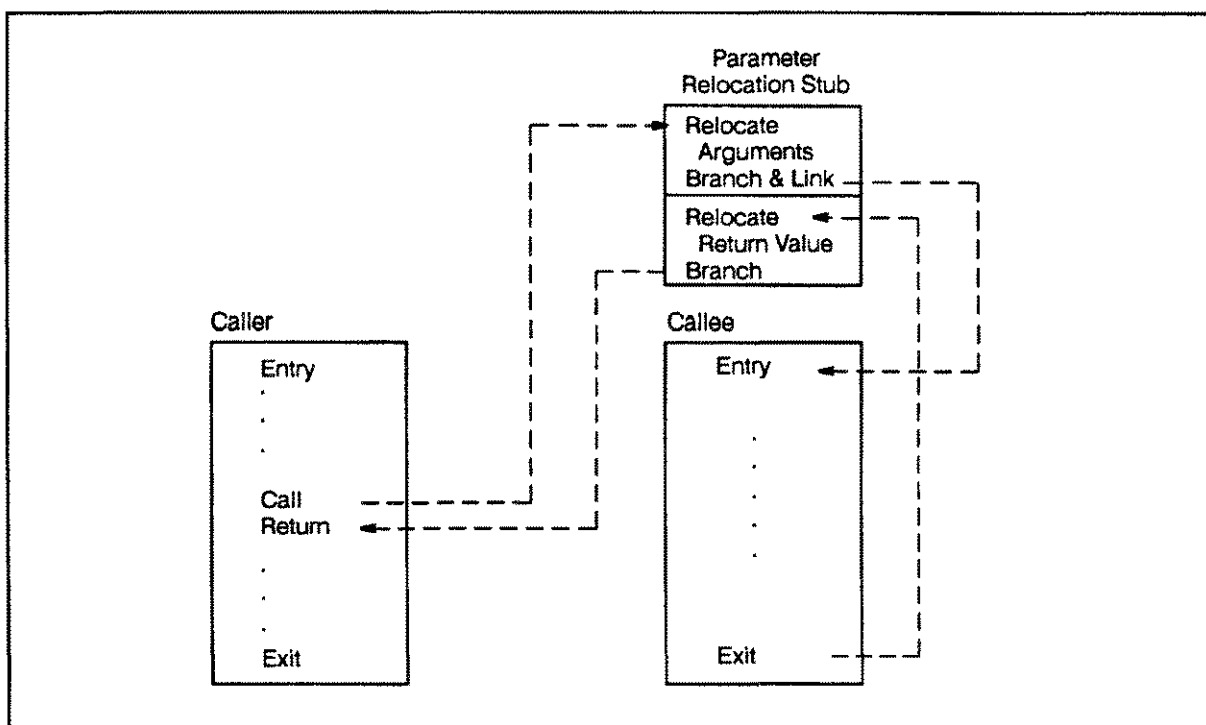
- 00            Do not relocate
- 01 arg        Argument register
- 10 fr         Floating-point register (bits 0..31)\*
- 11 frupper   Floating-point register (bits 32..63)\*

\* For return values, '10' means a single precision floating-point value, and '11' means double precision floating-point value.



When the linker resolves a procedure call, it will generate a relocation stub if the argument location bits of the fixup request do not exactly match the relocation bits of the exported symbol. One exception is where either the caller or callee specifies "do not relocate". The relocation stub will essentially be part of the called procedure, and the linker can optionally add a symbol record for the stub so that it can be reused. The symbol record will be the same as the original export symbol record, except that the relocation bits will reflect the input of the stub. The type will be *STUB* and the symbol value will be the location of the relocation stub.

The execution of a relocation stub can be separated into the call path and the return path. During the call path, only the first four words of the parameter list will be relocated, while only the function return will be relocated during the return path. The control flow is shown in Figure 3-3.



LG200010\_014

**Figure 3-3. Parameter Relocation Stub.**

If the function return does not need to be relocated, the return path can be omitted and the branch and link will be changed to a branch. The call path must always be executed, but if the first four words of the parameter list do not need to be relocated, it can be reduced to the code required to establish the return path (i.e. save RP and branch and link to the callee).

When multiple stubs occur during a single call (e.g. calling stub and relocation stub), the stubs can be cascaded (i.e. used sequentially); in such a case, both RP' and RP'' would be used. (The relocation stub uses RP'')

When the linker makes a load module executable, it will generate stubs for each procedure that can be called from another load module (i.e. called dynamically). In addition, a stub will be required for each possible calling sequence. Each of these stubs will contain the code

for both relocation and external return, and will be required to contain a symbol definition record.

Both calling and called stubs use a standard interface: calling stubs always relocate arguments to general registers, and called stubs always assume general registers.

In order to optimize stub generation, the compilers should maximize the use of the argument location value 00 (do not relocate). A linker option may be provided, which will allow the user to turn stub generation on or off, depending on known conditions. Also, a linker option is provided to allow the user to inhibit the generation of stubs for run-time linking. In this case, if a mismatch occurs, it will be treated as a parameter type checking error (which is totally independent of parameter relocation).

Assembly programmers can specify argument relocation information in the `.CALL` and `.EXPORT` assembler directives.

## The Actual Call

---

This chapter describes the sequence of steps involved in executing a standard procedure call on PA-RISC.

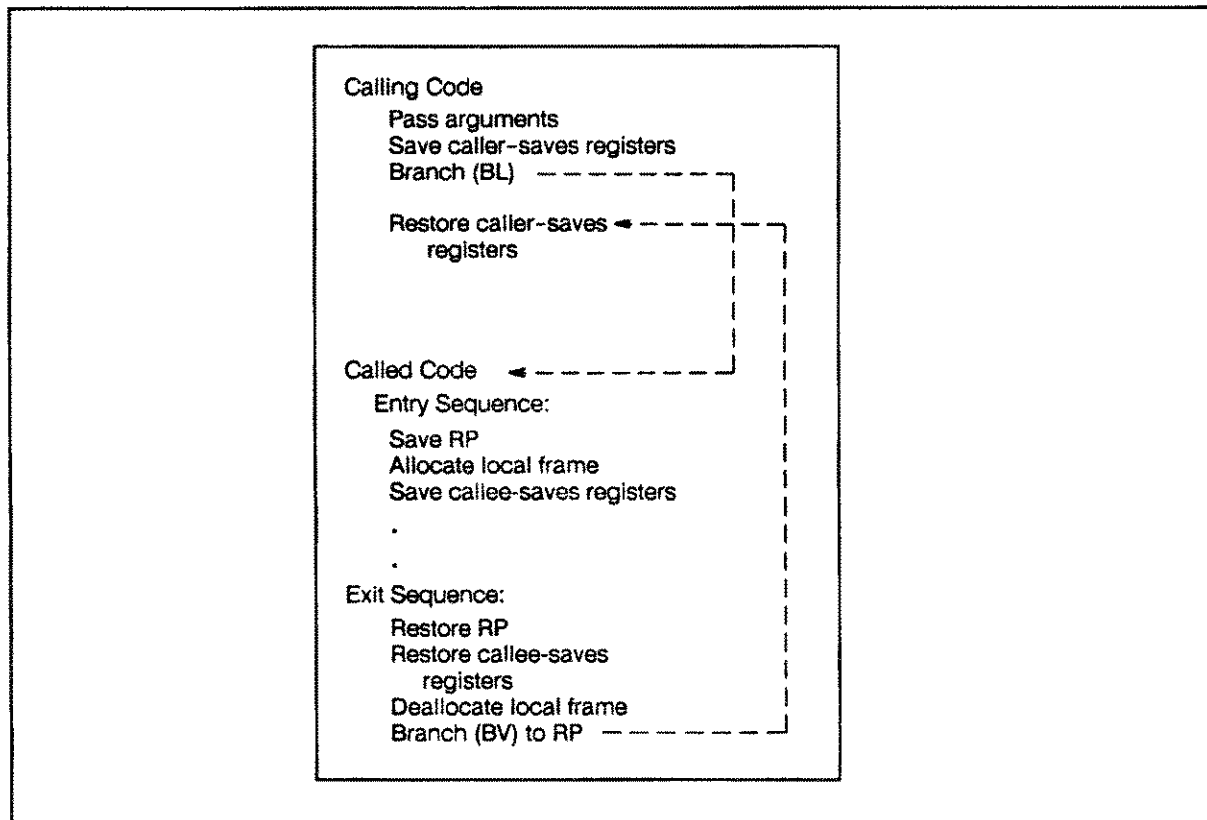
---

### 4.1. Control Flow of a Standard Procedure Call

Figure 4-1 shows the progression of a standard procedure call. To read the diagram, begin at “Calling Code” and continue downward, following any arrows that may extend from the end of a line.

To summarize, the steps involved are:

1. In the calling procedure, complete all modifications to variables that may be used by the procedure about to be called.
2. Before the call, place all actual arguments onto the stack or into argument registers. As necessary, save any values being held in caller-saves registers to memory.
3. Branch to the callee using a **BL** instruction with **RP** as the target (link) register.
4. Upon entry, save **RP** in the caller’s frame marker (if this is a non-leaf procedure). Allocate the local frame by incrementing the stack pointer, and save any callee-saves registers that will be used.
5. Complete execution of the body of the procedure.
6. Before exiting, restore **RP**, restore callee-saves registers, and deallocate local frame.
7. Branch back to the caller using a **BV** instruction.
8. Upon return to the caller, store the function result and restore caller-saves registers as necessary.
9. Continue execution.



LG200010\_015

**Figure 4-1. Control Flow of a Standard Procedure Call.**

## 4.2. Efficiency

The following factors greatly reduce the overhead expense involved in a PA-RISC procedure call:

1. Allocation of the stack frame and saving of one callee-saves register can be accomplished in a total of one cycle. (The same is also true for the deallocation of the frame and restoration of one register.)
2. Even when not optimizing, the delay slot (i.e. the instruction slot following a Branch instruction) is used for operations such as loading register values and passing parameters.
3. Most of the steps involved in a call are optional if they are unnecessary in a particular situation. (For example, saving and restoring RP, saving and restoring callee-saves registers, allocating stack storage space, etc.)

---

### 4.3. The Code Involved in a Simple Local Call

A simple example is shown below; the procedure and the call to it are shown first in C (source code) and then in assembly. A more detailed, fully documented example is shown in Appendix A.

In C:

```

:
:
proc (50,100);      /* call to procedure 'proc'      */
:
:
proc (x,y)         /* function 'proc'          */
int x,y;          /* returns the sum of two integers */
{
    return x+y;
}
:
:
```

In assembly:

```

:
LDI  50, gr26      ; load arg word 0 into gr26
** BL  proc, gr2   ; branch (call) to 'proc'
LDI  100, gr25     ; <delay slot> load arg word 1 into gr25
:
:
proc: BV  0 (gr2)   ; branch (return) back to caller
      ADD  gr26,gr25,gr28 ; <delay slot> add arguments, put result into gr28
:
:
```

\*\* In instances where the target of the call is out of the range of the BL (i.e. greater than 256K bytes), there will be a slightly different code sequence. If this situation is identified at compile time, the compiler will generate a three-instruction sequence (LDIL,BLE,COPY) that is guaranteed to reach the target, instead of the BL. However, if the information is not known until link time (the compiler has already generated the BL), the BL will be linked to a "long branch stub" rather than to the actual procedure. This "stub" is merely the same two-instruction sequence that would have been used if the information had been known earlier.



## Inter-Module Procedure Calls

---

External calls exist on both the MPE XL and HP-UX operating systems. Although the external call mechanism is similar on the two operating systems, the details of the call mechanism differ enough to warrant separate discussions. The description of external calls on MPE XL and the details of HP-UX external calls are both described in this chapter.

---

### 5.1. External Calls on MPE XL

As mentioned earlier, there are three types of procedure calls that can be executed, and they can be classified into two groups: intra-module (local) and inter-module (external) calls. Basically, a local call is one in which the caller and the callee reside in the same load module, and an external call is one in which that is not necessarily the case. There is one exception to this definition, however: calls which cross privilege level boundaries are always treated as external calls, even if the caller and callee reside in the same load module. Although external calls are closely related to local calls, several notable differences exist in storage and access conventions; these differences are explained in the following material.

The inter-module (external) calling sequence is distinguished from the intra-module (local) calling sequence to take advantage of the system-wide global virtual addressing of the PA-RISC processor and to implement code sharing. Unlike most conventional systems where each process has a private virtual address space, all processes in a PA-RISC system share a finite, although large, number of spaces ( $2^{16}$  currently,  $2^{32}$  potentially). Therefore, it is undesirable to assign virtual spaces to inactive program images (i.e. those on disk). Generally, the assignments of virtual spaces to a program will be delayed until the program is activated.

In order to avoid extensive linking at process creation time, it is desirable to have a central data structure (for each process) that will hold the SIDs assigned to the load modules used by the process. This data structure is the XRT (Inter-Module Cross Reference Table). Only the entries in the XRT need to be updated when virtual spaces are finally assigned to load modules.

Another use for the XRT is the location of the global data area of a load module. The offset of the global data area in the private data space of a process depends on the combination of load modules called by the process. Although each load module's code is assigned a unique SID, its data is placed at a process-dependent offset within the process' single private data space. In order to share a load module between multiple processes, all references to the global data area must be relative to the base register, DP (data pointer). The value of DP is stored in a load module's entry in the XRT. The XRT will be discussed and diagramed in detail below.

An external call uses the same code sequence as a local call, with the addition of a "calling stub" (a.k.a. Import Stub) attached to the calling code, and a "called stub" (a.k.a. Export Stub) attached to the called code. Unlike a local call, execution is not transferred directly from the caller to the callee; in an external call, a millicode sequence (CALLX) is employed

to locate and branch to the target procedure. These stubs and the millicode sequence are discussed in greater detail below.

### **5.1.1. Requirements of an External Call on MPE XL**

An external procedure call requires several extra steps in addition to those necessary for a local call, as outlined below:

1. The base register pointing to the global data area (DP), the SID contained in sr4, and value in gr2 (RP) must all be saved in the caller's allocated frame area. This RP value is referred to as RP' to distinguish it from the usual RP value associated with a local call. The difference between RP and RP' is as follows:  
  
RP is renamed RP' during the calling stub execution, at which time RP becomes the location in the called stub to which the target procedure (callee) must return. Finally, RP' is renamed back to RP, and DP and the SID are restored.
2. The called load module's entry in the XRT must be located.
3. The short pointer from the XRT entry must be loaded into DP (this is the DP value for the called load module).
4. The SID from the XRT entry must be loaded into sr4, and the offset of the callee must be obtained from the XRT.
5. An external branch and link to the called procedure must be performed. (Actually, an external branch to the CALLX millicode routine that locates the called stub and a branch to the called stub of the callee, which then does a local branch and link to the callee. Stubs, CALLX, and linking will be discussed further in the section, Import/Export Stubs.)

### **5.1.2. Requirements of an External Return**

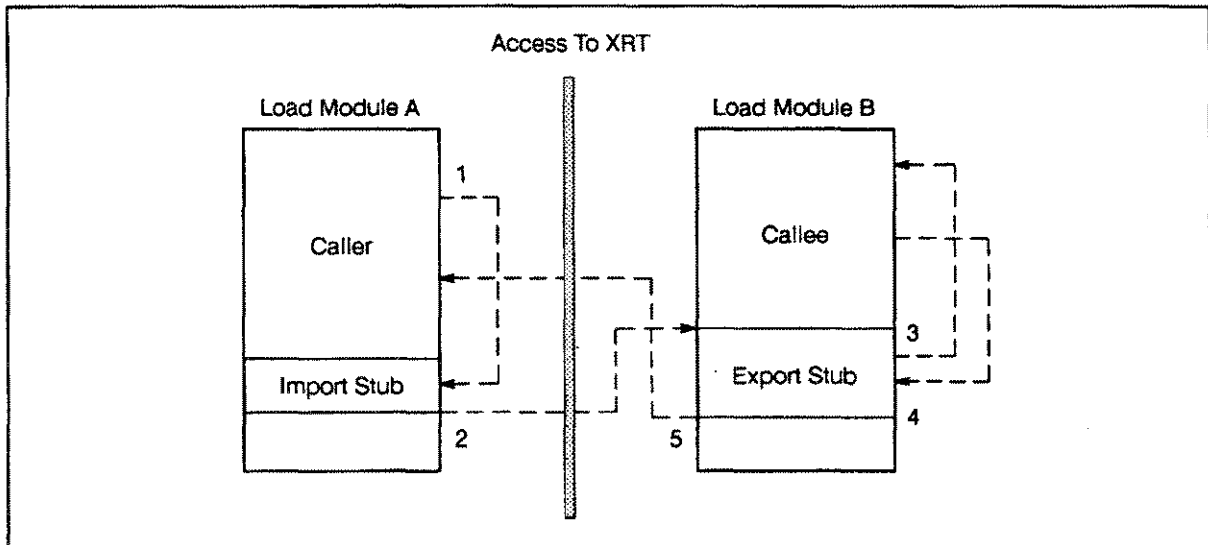
In order to return from an external call, it is required that DP, sr4, and RP be restored. The saved value of RP' will be loaded into RP (gr2), and the return to the caller will be via that register. The DP and SID of the caller will be restored from the caller-save area.



### 5.1.3. Control Flow of an MPE XL External Call

Figure 5-1 shows a simplified external procedure call. It uses the same code sequence as the local, but a “calling stub” is attached to the calling code (the caller) and a “called stub” is attached to the entry point of the called code (the callee). Execution is transferred from the calling code to the called code by executing a millicode sequence (CALLX) which uses an XRT to locate and branch to the target procedure. Note: All of these elements of an external call are covered in more detail in the following subsections.

To read the diagram, follow the arrows and numbers, beginning with number 1.



LG200010\_016

Figure 5-1. Simplified External Procedure Call.

### 5.1.4. Calling Code

The calling code generated by the compiler to perform a procedure call will be the same regardless of whether the call is local or external. If the linker locates the procedure being called within the current executable load module, it will make the call local by patching the BL instruction to directly reference the entry point of the procedure. If the linker is unable to locate the procedure, it will make the call external by attaching a calling stub to the calling code, and patch the BL instruction to branch to the stub.

Before the call, the calling code must save any caller-saves registers that contain active data. The parameter list for the callee is stored in the current stack frame between the register spill area and the frame marker. As in a local call, the parameter list is stored in reverse order, such that the first word is at SP-36, the second is at SP-40, etc. Note: the first four words of the parameter list are passed in registers, but the space for the argument list is allocated in the stack frame, even if the first four words are unused. Also, by default all parameters are passed in general registers, with the linker including any necessary relocation stubs. See Chapter 3, Parameter Relocation.

### 5.1.5. Called Code

The called code is responsible for allocating a new stack frame on the top of the stack (the frame must be 64-byte aligned); the actual size of the frame will be determined by the compiler, and will be the summation of:

- The amount of space needed by the register allocator for the register spill area;
- The amount of space needed for the local variables of the current procedure;
- The amount of space needed to store the longest parameter list of any procedure called by this procedure; and
- The 32-byte frame marker.

If this procedure is callable by a less-privileged procedure, each page of the stack frame must be PROBE'd (a privilege-checking mechanism) before any information is stored into the frame. The PROBE instructions must be generated by the compiler. (This is not currently implemented).

When a procedure is entered, gr2 (RP) will contain the offset portion of the return address. Whether the procedure was called locally or externally, the two low order bits of RP will always contain the execution level of the caller. From the source code level it is difficult, if not impossible, to determine if RP is valid or if it has been stored into the stack frame. Therefore, compilers that support multiple privilege levels need to provide a mechanism for returning the execution level of the caller.

If the current procedure calls another procedure, RP must be saved sometime before the call, usually in the procedure entry sequence. The called code is responsible for insuring the validity of its own input. sr5, sr6, and sr7 can only be set by privileged code and, therefore, can be assumed to be correct at all times. In addition, DP, LP (linkage pointer), and sr4 are not changed during a local call, or they are set by the procedure call millicode during an external call and therefore may also be assumed to be correct. The value of SP and the parameters passed must be validated by the called code.

Only those fields necessary in the frame marker of the current procedure will contain valid data; others will be undefined. For example, during a local call, contents of the external return link pointer field will be undefined.

### 5.1.6. Import and Export Stubs

As previously mentioned, the compiler only generates a single type of procedure call (local), a characteristic that is made possible by the use of stubs. Stubs are pieces of code that are attached to the caller and/or callee that enable the original calling and called code to remain unchanged through the external call process. There are two types of stubs used in this procedure calling convention: Import (calling) and Export (called). These are defined here, and explained in detail in the following sections.

1. Import Stub (Calling Stub)—a locally-linked stub that enables inter-module and OS/Subsystem calls to appear (to the compiler) as local calls. If the linker determines that a call is external, it will attach a calling stub to the procedure and will patch the BL (Branch and Link) instruction to branch to the stub. There is usually one calling stub for each procedure (and privilege level combination) referenced in the module (which can accommodate all calls to a specific procedure), but it is possible to have a separate stub for each CALL to a procedure.

2. Export Stub (Called Stub)—enables a called routine to avoid the problem of having multiple return sequences (i.e. different for local and external calls). There is one called stub for each external procedure of a load module. Inter-module calls will enter the called stub, which in turn will enter the called procedure (callee). Thus, the callee can return to its called stub (which is local) rather than being concerned with the external return. The calling stubs can be generated by the linker (or obtained from a “stub library”) and then linked to their respective routines.

### 5.1.7. Import Stubs

The import stub will load gr1 with a pointer to the procedure XRT table entry (XRT pointer) of the called procedure and then branch to an external procedure call millicode sequence. Since the location of the XRT for a load module may be different for separate executions of the load module, the XRT entry pointer will be computed in the import stub. The XRT entry pointer is computed by adding the XRT entry offset to the value of the LP, which is stored at DP-4, pointing to the base of the XRT for the current load module.

For permanently bound calls to the operating system, an import stub is not necessary; instead, the BL instruction in the call is replaced with a BLE instruction that branches to a system entry point branch table. This eliminates much of the linking that is normally performed when a load module is loaded. (This is currently not implemented.)

Although the external procedure call diagram (Figure 5-5) shows that DP, RP', and sr4 are saved by the CALLX millicode (see next section for discussion of CALLX), DP and RP' will actually be saved in the import stub, and sr4 will be copied to a general register in the import stub. This is done to eliminate two interlocks and fill a branch delay slot that would otherwise be left unused. The code sequence of the calling stub will be similar to that shown below:

```
LDW      -4(DP),gr1      ; Load LP
STW      DP,-32(SP)     ; Save DP
ADDIL *  L'XRToff,gr1   ; Add XRT offset to LP
LDO *    R'XRToff(gr1),gr1 ;
LDW      16(gr1),gr20   ; Load address of CALLX
STW      RP,-24(SP)    ; Save RP'
BE       (sr7,gr20)     ; Branch to CALLX
MFSP     sr4,gr21      ; Move sr4 to gr21
```

- \* Can be eliminated by the linker in cases where they would effectively be NOPs. In these cases the total size of the stub is padded to 8 words because unwind descriptors assume fixed-length stubs.

### 5.1.8. External Procedure Call Millicode (CALLX)

The CALLX millicode sequence is primarily a transition mechanism that facilitates the successful location and access of the desired external routine. The address of the CALLX routine is obtained from the XRT entry, and is assigned by the loader. Seven variations of CALLX are available, depending on the possible privilege promotions. CALLX is called from the calling stub, and operates as follows:

1. Saves DP, RP', and sr4 (if necessary).
2. Alters the privilege level if necessary (Gateway).
3. Checks the XRT pointer to insure that it points to a valid XRT table entry.
4. Loads the LP, DP, Offset and sr4 (of new procedure) values.
5. Branches to the called stub in the external module.

### 5.1.9. Export Stubs

An export stub is used to allow the compiler to generate the same exit code sequence regardless of whether the procedure will be called locally or externally. If the linker determines that a procedure can be called from another load module, it will attach a called stub to the procedure. The stub will be the external entry and return point for the procedure. Local calls to the procedure will be unaffected.

Although the stub is the external entry point, its primary purpose is to be executed during an external procedure call exit/return. The stub is entered before the procedure so that RP can be set to the address of the stub, which will cause the local return in the procedure to exit to the stub. When the stub is executed during the return, it will restore DP and sr4, and return to the caller.

The stub executes at the caller's execution level.

The code sequence for the called stub will be similar to the following:

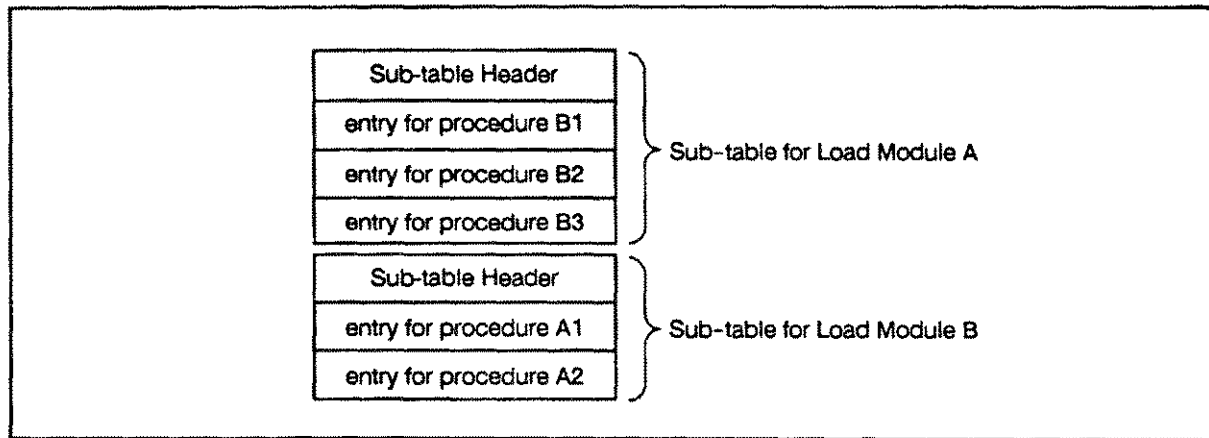
```
BL      disp,gr2           ; Branch to local entry point
DEP     gr31,31,2,gr2     ; Deposit caller's Exec. Level in link
LDW     -28(SP),gr21      ; Restore sr4 (part 1)
LDW     -24(SP),gr2      ; Restore return address (RP')
MTSP    gr21,sr4         ; Restore sr4 (part 2)
BE      0(sr4,gr2)       ; Branch back to caller
LDW     -32(SP),gr27     ; Restore DP
```

### 5.1.10. Inter-Module Cross Reference Table (XRT)

The Cross Reference Table (XRT) is used to link the external procedure calls of a load module. Every process has an XRT area reserved from its process space (pointed to by sr5). The table contains a sub-table for each load module referenced during process execution. Each sub-table for a load module contains entries for all the procedures called by that load module. A sample XRT is shown in Figure 5-2 (in this example, the process has two load modules: 'A' and 'B'. 'A' calls procedures B1, B2 and B3; 'B' calls procedures A1 and A2).

### 5.1.11. The Layout of the XRT

One XRT might be visualized as shown in Figure 5-2. (This diagram corresponds to the calling situation described in the last sentence of the previous section.)



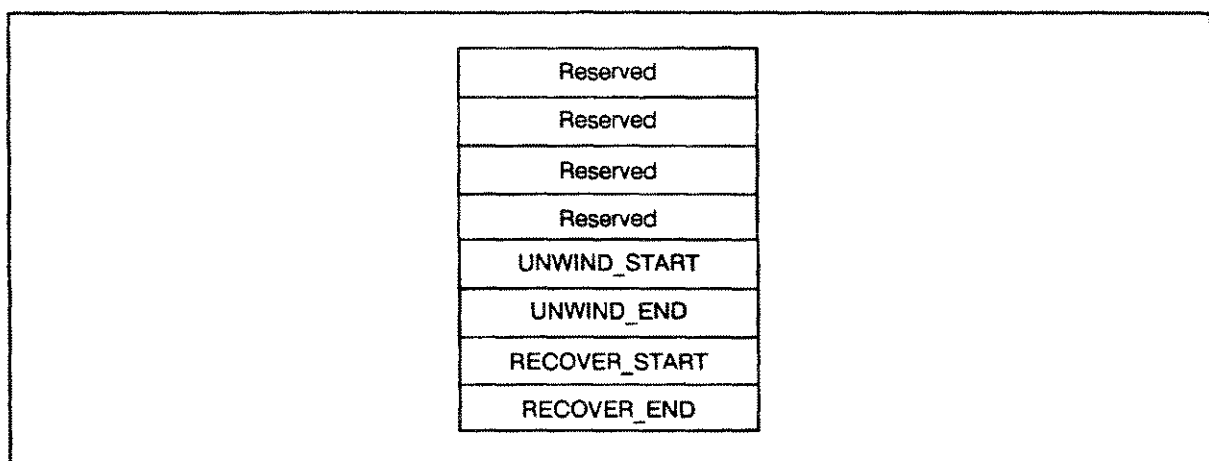
LG200010\_017

Figure 5-2. Layout of the XRT.

### 5.1.12. Sub-table Header

Each sub-table of a load module in the XRT contains an eight word header used to locate unwind tables for the module. See Chapter 7, Stack Unwinding. The sub-table header, as shown in Figure 5-3, contains the following information:

- (The first four words are presently undefined and are reserved for future use.)
- The starting address of the Unwind table.
- The starting address of the Linker Stub Unwind table.
- The starting address of the Recover table.
- The starting address of the Auxiliary Unwind table.



LG200010\_025

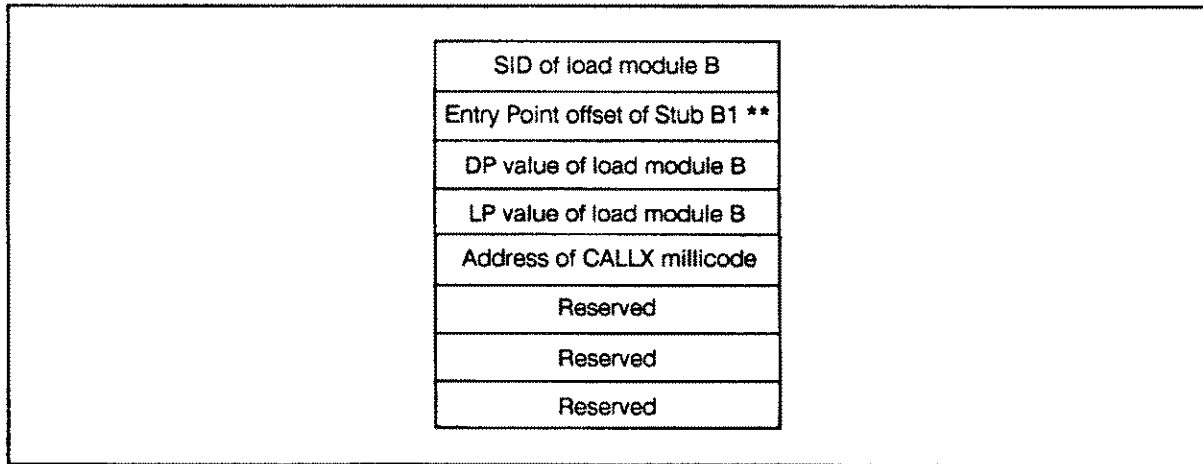
Figure 5-3. Sub-table Header.

If a load module contains no external references, its sub-table in the XRT will contain only the header.

An entry for a procedure within a sub-table of a load module in the XRT (e.g. the entry for B1) is eight words long, and contains the following information:

1. The SID of the module to which it belongs.
  2. The entry offset for the procedure. This is a 32-bit offset, and is the address of the entry point (relative to the base of the SID of its load module) of the procedure's called stub. The last two bits (30 and 31) of this word must be zero in order to insure word alignment of the address.
  3. The DP value for the load module to which it belongs (the value of the base register pointing to the load module's global data area).
  4. The LP value of the module in which the called procedure is contained. This is a pointer to the beginning of the XRT sub-table of that load module.
  5. The address of the CALLX millicode routine.
- 6.,7.,8. (These three words are presently undefined and are reserved for future use.)

The XRT entry for procedure B1 (which is called from A and would appear in A's sub-table of the XRT) is shown in Figure 5-4.



LG200010\_026

**Figure 5-4. One XRT Entry.**

\*\* The last two bits of the Entry Point Offset must be set to zero in order to ensure word alignment.

### 5.1.13. Linkage Pointer

A single value, the Linkage Pointer, resides in the word directly below the global data area of a load module, at the location pointed to by DP-4. This pointer is private to the load module, and is a short pointer to the beginning of the load module's XRT sub-table. An entry for a called procedure in the XRT is pointed to as follows:

1. The LP points to the beginning of the sub-table in the XRT of the load module containing the called procedure.
2. The import stub for the caller has the offset to the called procedure's entry relative to the XRT sub-table of the caller's load module. This offset, added to the LP value, provides a pointer to the called procedure's entry. This LP-relative XRT offset is assigned by the linker.
3. The reason for the indirection employed by using the LP is that load modules can be shared by different processes whose XRTs may also be different. To allow the same code to reference the same load module in different processes' XRTs, it is necessary to provide a uniform interface to the XRT entries; this is provided by the LP.

In addition to the XRT area in the process space, there is an XRT area in the system space (pointed to by sr7) that is reserved for the XRTs of system load modules. Like any other load module, a system load module also uses LP to locate its XRT. The system XRT area can also contain a special XRT that is used for calling system procedures by intrinsic number.

### 5.1.14. System Security and the XRT

The XRT will be set up by the loader. The values in the XRT will be supplied by the loader, based on mapping the files relevant to the process into virtual memory (i.e. SID allocation, the data offsets in sr5 space, etc.). The linker may provide some of the values that are to be contained in sr5, based on the information it may have at link time concerning the specific load modules that are involved in the process' executable image.

When a process is loaded, the loader will protect all the pages in the XRT to read level 3, write level 0. Although it is not necessary, the process protection ID will be assigned to the pages of the process XRT area. Since the LP is located with the process' global data area, its protection is the same as that load module's global data area. It is not necessary to validate LP because it will be set by CALLX at every external call or privilege level change.

The XRT of every process and the system XRT must be at the same offset of their corresponding quadrant, and every XRT must be the same length. These two restrictions allow the procedure call millicode (CALLX) to use a very simple masking algorithm to perform bounds checking on any XRT pointer used with an external call. The location and size of the area can be changed when the system is restarted, but the new values must be reflected in the procedure call millicode (because it uses constant values to do bounds checking on the XRT entry pointer).

### 5.1.15. Interface Between Import and Export Stubs

The exact distribution of all operations between the import and export stubs, and whether the export stub uses a centralized system routine to accomplish these tasks, is not architected. Much more important, and specified in detail, is the work that the export stub expects to have been done before it is entered. Adhering to these requirements facilitates the loading of DP, LP, and SID (if desired) of the called load module. These requirements are as follows:

1. gr1 must contain a pointer to the called procedure's XRT entry. (This is actually the called procedure's XRT entry in the sub-table of the calling load module.)

Recall that the caller's LP points to the caller's XRT sub-table, which contains entries for all of the routines that may be called. The offset into that sub-table, which indexes to the called routine's entry, is bound as an immediate in the import stub. The pointer to the specific XRT entry is calculated as follows:

(caller's LP value) + (offset to called procedure's entry) = (pointer to callee's entry in XRT sub-table of caller)

This pointer is the value that should be found in gr1 when the export stub is entered.

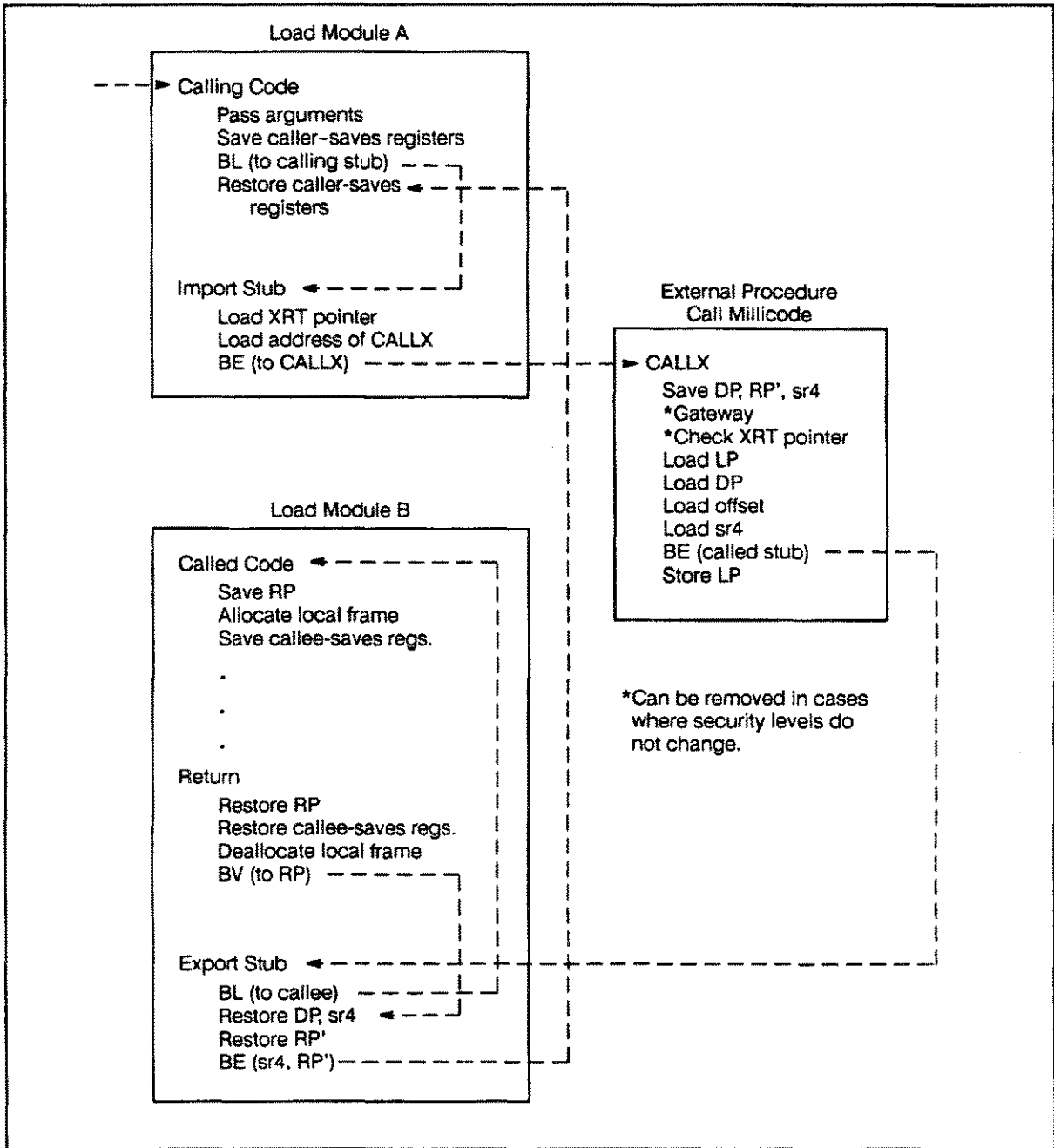
2. The SID of the called load module must be loaded into sr4. (The export stub is free to check the validity of that SID, to reload it, or to leave it as is. The important point is the assumption that it has already been loaded, and DOES NOT need to be checked.)

### 5.1.16. Summary of an External Procedure Call

Figure 5-5 shows a detailed picture of the flow of control associated with an external procedure call.

To read the diagram, begin at the upper left-hand corner ("Calling Code"), and read downward; whenever an arrow extends from a line, follow it, and continue downward in the box from the point where the arrow ends.





LG200010\_018

Figure 5-5. Summary of External Procedure Call.

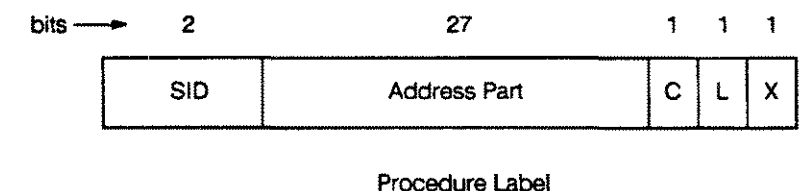
### 5.1.17. Dynamic Linking

Dynamic linking is the process of run-time linking to routines. Dynamic linking is required when the target procedure is unknown at compile time, or the target of a procedure call can change while the code is executing. Dynamic linking is carried out through explicit protocols (e.g. the HPGETPROCPLABEL intrinsic on MPE XL).

If the dynamically linked routine resides in a load module that has not yet been loaded, the load module is loaded dynamically. In order to dynamically load a load module, a global data area for it may need to be allocated in sr5 space. This data space is allocated by the loader, and may be allocated from any unused virtual space in sr5.

### 5.1.18. Procedure Labels

A procedure label is a specially-formatted variable that is used to link dynamic procedure calls. The format of a procedure label is shown below:



LG200010\_027

The X field in the address section of the procedure label is the XRT flag, which is used by compilers to determine if the procedure label is local (off) or external (on). In the case of a local procedure label, the address part will be a pointer to the entry point of the procedure, while in the external case, the address part will be a pointer to an XRT entry for the procedure.

For external procedure labels, the linker supplies an LP offset. Generated code must optionally add LP to this value to produce an XRT address.

The L field in the address part of the procedure label is the shared library flag (used only on HP-UX).

The C field is mentioned for completeness. It is only relevant when the X field is on and it is used to indicate a Compatibility Mode procedure label. CM plabels are never called directly from native code.

The following is an example of the code generated by compilers to obtain the address of a procedure (a procedure plabel):

```
LDIL    L'func,1      ; get the address of the function
LDO     R'func(1),31
EXTRU,= 31,31,1,19   ; check the X-field to determine if XRT address
LDW     -4(0,27),19  ; if it's an XRT address
ADD     31,19,20     ; add in LP
```

In the current implementation on MPE XL, the L-field is never turned “on” and is, therefore, effectively unused. In the future, either the specification or the implementation may change to use this field.

## 5-12 Inter-Module Procedure Calls

The dynamic procedure call millicode, *\$\$dyncall*, determines if a procedure label is local or external, and takes the appropriate action. (A local procedure label can only be used to call procedures within the current load module.) The following pseudo-code sequence demonstrates the process used for dynamic calls. Note the similarity between this sequence and the import stub sequence:

```
IF (X-field in Plabel) = 0 THEN
    Branch Vektored using Plabel
ELSE BEGIN
    Clear X-field;
    Save DP;
    Load address of CALLX;
    Save RP';
    Move sr4 to gr21;
    Branch to CALLX;
END.
```

The X and L flags must be zero during an external call, or they will cause a misaligned data reference trap when accessing the XRT. (As mentioned earlier, the L flag is currently unused on MPE XL, so it is assumed to be zero.)

An external procedure label can be used in conjunction with the external procedure call millicode to call any procedure within the process or the operating system (subject to XLeast checking to insure adequate execution level). The procedure call millicode only uses the address part of the procedure label, but it may point to either the process or system XRT.

The intrinsic HPGETPROCPLABEL may be used to get an external procedure label for any level 1 procedure in a process. If the compiler or linker determines the need for an external label, it is communicated to the loader by a normal import request or an explicit call to HPGETPROCPLABEL.

Although a procedure label pointing to a system XRT entry is valid for all processes, it will be unloaded when its reference count drops to zero. Therefore, these procedure labels should not be considered as global procedure labels. The procedure HPGETSYSPLABEL will return a global procedure label for any procedure in a system load module, but it requires privilege level 1 to be called.

---

## 5.2. External Calls on HP-UX

External calls occur on HP-UX, in both shared libraries and the programs which use them. A shared library contains subroutines that are shared by all programs that use them. Shared libraries are attached to the program at run time rather than copied into the program by the linker. Since the shared library code is not copied into the program file and is shared among several programs as a separate load module, an external call mechanism is needed.

In order for the object code in a shared library to be fully sharable, it must be compiled and linked in such a way that it does not depend on its position in the virtual addressing space of any particular process. In other words, the same physical copy of the code must work correctly in each process.

Position independence is achieved by two mechanisms. First, PC-relative addressing is used wherever possible for branches within modules and for accesses to literal data. Second,

indirect addressing through a per-process linkage table is used for all accesses to global variables, for inter-module procedure calls and other branches and literal accesses where PC-relative addressing cannot be used. Global variables must be accessed indirectly since they may be allocated in the main program's address space, and even the relative position of the global variables may vary from one process to another.

Position-independent code (PIC) implies that the object code contains no absolute addresses. Such code can be loaded at any address without relocation, and can be shared by several processes whose data segments are allocated uniquely. This requirement extends to DP-relative references to data (these are considered absolute, since the DP register is essentially an absolute constant on HP-UX). In position-independent code all references to code and data must be either PC-relative or indirect. All indirect references are collected in a single linkage table that can be initialized on a per-process basis.

The Linkage Table (LT) itself is addressed in a position-independent manner by using a dedicated register, gr19, as a pointer to the Linkage Table. The linker generates import (calling) and export (called) stubs which set gr19 to the Linkage Table pointer value for the target routine, and handle the inter-space calls needed to branch between shared libraries.

The code in the program file itself does not need to be position independent, but it must access all external procedures through its own linkage table by using import stubs. The Linkage Table in shared libraries is accessed using a dedicated Linkage Table pointer (LTP), whereas the program file accesses the Linkage Table through the DP register.

Position independent code is generated by the compilers when the +z or +Z compilation flag is used. Code which is used in a shared library must be compiled with this flag. Code in the program file is compiled without this flag and the linker places the import/export stubs into the program file to handle calls to and from files which are external to the program load module.

### **5.2.1. Control Flow of an HP-UX External Call**

The code generated by the compiler to perform a procedure call is the same whether the call is external or local. If the linker locates the procedure being called within the program file, it will make the call local by patching the BL instruction to directly reference the entry point of the procedure. If the linker determines that the called procedure is outside of the program file, it makes the call external by inserting an import stub (calling stub) into the calling code, and patching the BL instruction to branch to the stub. For any routine in the program file which the linker detects is called from outside of that program file, an export stub (called stub) is inserted into the program file's code.

When building a shared library (with the linker's ld -b flag), the linker generates import and export stubs for all procedures which can be called from outside of the shared library.

Figure 5-6 below shows the control flow of an external call on HP-UX.

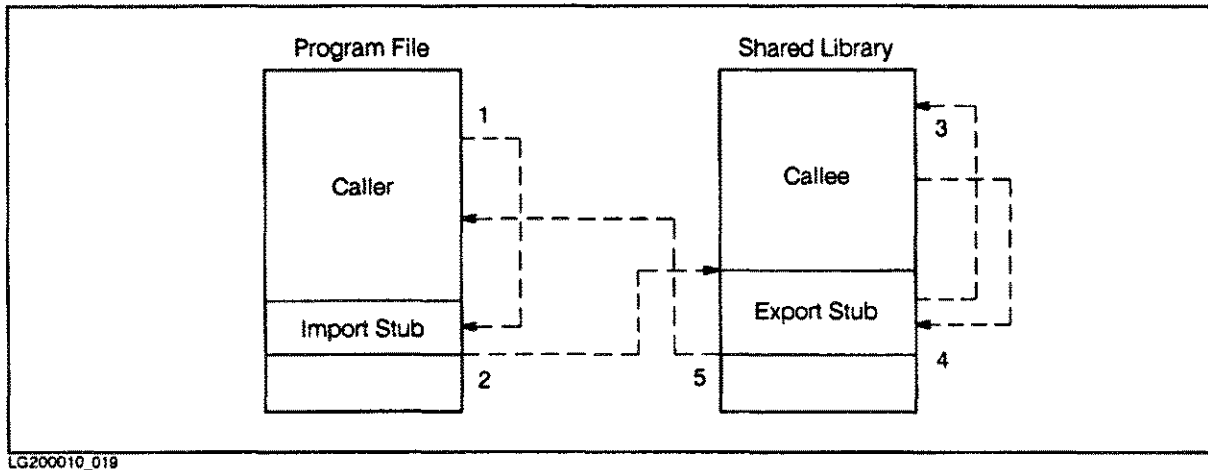


Figure 5-6. HP-UX External Procedure Call.

### 5.2.2. Calling Code

The calling code in program files is responsible for performing the standard procedure call steps regardless of whether the call is external or local. The linker generates an import stub to perform the additional steps required for external calls.

The import stub (calling stub) of an HP-UX external call performs the following steps:

1. Loads the target (export stub) address of the procedure from the Linkage Table
2. Loads into gr19 the LTP (Linkage Table Pointer) value of the target load module.
3. Saves the return pointer (RP'), since the export stub will overwrite RP with the return address into the export stub itself.
4. Performs the interspace branch to the target export stub.

The code sequence of the import stub used in the program file is shown below:

```

;Import Stub (Program file)

X':    ADDIL    L'lt_ptr+ltoff,dp      ; load procedure entry point
        LDW     R'lt_ptr+ltoff(1),r21
        LDW     R'lt_ptr+ltoff+4(1),r19 ; load new gr19 value.
        LDSID   (r21),r1              ; load space id of proc. entry
        MTSP    r1,sr0                ; move space id from general to
                                       ; space register
        BE     0(sr0,r21)             ; branch to target
        STW     rp,-24(sp)            ; save RP'

```

The difference between a shared library and program file import stub is that the Linkage Table is accessed using gr19 (the LTP) in a shared library, and is accessed using DP in the program file.

The code sequence of the import stub (calling stub) used in a shared library is shown below:

```
;Import Stub (Shared Library)

X':   ADDIL   L'ltoff,r19      ; load target (export stub) address
      LDW    R'ltoff(r1),r21
      LDW    R'ltoff+4(r1),r19 ; load new gr19 (LTP) value
      LDSID  (r21),r1         ; load space id of target address
      MTSP   r1,sr0          ; move space id to space register
      BE    0(sr0,r21)       ; branch to target
      STW   rp,-24(sp)       ; save RP'
```

### 5.2.3. Called Code

The called code in shared library files is responsible for performing the standard procedure call steps regardless of whether the call is external or local.

The linker generates an export stub to perform the additional steps required for shared library external calls. The export stub is used to trap the return from the procedure and perform the steps necessary for an inter-space branch.

The export stub (called stub) of a shared library external call performs the following steps:

1. Branches to the target procedure. The value stored in RP at this point is the return point into the export stub.
2. Upon return from the procedure, restores the return pointer (RP').
3. Performs an interspace branch to return to the caller.

The code sequence of the export stub is shown below:

```
X':   BL,N    X,rp           ; trap the return
      NOP
      LDW    -24(sp),rp      ; restore the original RP
      LDSID  (rp),r1         ; load space id of return address
      MTSP   r1,sr0          ; move space id from general reg. to space reg.
      BE,N   0(sr0,rp)       ; inter-space return
```

---

## 5.3. PIC Requirements for Compilers and Assembly Code

Any code which is PIC or which makes calls to PIC must follow the standard procedure call mechanism. In addition, register gr19 (the linkage table pointer register) must be stored at sp-32 by all PIC routines. This should be done once upon procedure entry. Register gr19 must also be restored upon return from each procedure call, even if gr19 is not referenced explicitly before the next procedure call. The LTP register, gr19, is used by the import stubs and must be valid at all procedure call points in position independent code. If the PIC routine makes several procedure calls, it may be wise to copy gr19 into a callee-saves register as well, to avoid a memory reference when restoring gr19 upon return from each procedure call. As with gr27 (DP), the compilers must treat gr19 as a reserved register whenever position-independent code is being generated.

### 5.3.1. Long Calls

Normally, the compilers generate a single-instruction call sequence using the BL instruction. The compilers will generate a long call sequence if the user explicitly requests long branch generation via a command-line option, or if the module is so large that the BL is not guaranteed to reach the beginning of the subspace (where a stub can be inserted by the linker). The existing long call sequence is three instructions, using an absolute target address:

```
LDIL    L'target,r1      ; load target address into r1
BLE     R'target(sr4,r1); branch to target address
COPY    r31,rp          ; copy return address (link register) into rp
```

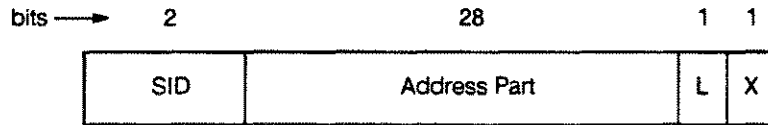
When the PIC option is in effect, the following seven-instruction sequence, which is PC-relative, must be used:

```
BL      .+8,rp          ; get PC into RP
ADDIL   L'target - $L0 + 4, rp ; add PC-rel offset to RP
LDO     R'target - $L1 + 8(r1), r1
$L0:    LDSID (r1), r31
$L1:    MTSP    r31, sr0
BLE     0(sr0,r1)
COPY    r31,rp
```

### 5.3.2. Procedure Labels and Dynamic Calls

The HP PA-RISC compilers generate the code sequence required for proper handling of procedure labels and dynamic procedure calls. Assembler programmers must use the same code sequence, described below, in order to insure proper handling of procedure labels and dynamic procedure calls.

A procedure label is a specially-formatted variable that is used to link dynamic procedure calls. The format of a procedure label is shown below:



Procedure Label

LG200010\_028

The X field in the address section of the procedure label is the XRT flag, which is used for MPE XL procedure labels to determine whether the call is local (off) or external (on). On HP-UX the L field is used to flag whether the procedure label is a pointer to an LT entry (L-field is on) or to the entry point of the procedure.

The plabel calculation produced by the compilers in both shared libraries and incomplete executables is modified by the linker, when building shared libraries and incomplete executables, to load the contents of an LT entry which is built for each symbol associated with a CODE\_PLABEL fixup.

In shared libraries and incomplete executables, a plabel value is the address of a PLT (Procedure Linkage Table) entry for the target routine, rather than a procedure address; therefore a utility routine named *\$\$dyncall* must be used when calling a routine via a procedure label. The linker sets the L field (second-to-last bit) in the procedure label to flag this as a special PLT procedure label. The *\$\$dyncall* routine checks this field to determine which type of procedure label has been passed, and calls the target procedure accordingly. The *\$\$dyncall* routine assumes that the X field is always 0.

The following pseudo-code sequence shows the process used by *\$\$dyncall* to perform dynamic calls:

```
IF (L-field in Plabel) = 0 THEN
    Perform interspace branch using Plabel as target address;
ELSE BEGIN
    Clear L-field;
    Load new LTP value into gr19;
    Load address of target;
    Save RP';
    Perform interspace branch to target address;
END.
```

In order to generate a procedure label that can be used for shared libraries and incomplete executables, assembly code must specify that a procedure address is being taken (and that a



plabel is wanted) by using the P' assembler fixup mode. For example, to generate an assembly plabel, the following sequence must be used:

```
; Take the address of a function
LDIL    LP'function,r1
LDO     RP'function(r1), r22
```

This code sequence will generate the necessary PLABEL fixups that the linker needs in order to generate the proper procedure label. The *\$\$dyncall* millicode routine in */lib/milli.a* must be used to call a procedure using this type of procedure label (i.e. a BL/BV will not work). For example:

```
; Now to call the routine using a plabel
BL      $$dyncall, 31 ; r22 is the input register for $$dyncall
COPY    r31, r2
```

The compilers generate the necessary code sequence required for proper handling of procedure labels.



## Millicode Calls

---

In a complex instruction set computer, it is relatively easy at system design time to make frequent additions to the instruction set based almost solely on the desire to achieve a specific performance enhancement, and the presence of microcode easily facilitates such developments. In a reduced instruction set computer, however, this microcode has been eliminated because it has been shown to be potentially detrimental to overall system performance (not only is instruction decode complicated, but the basic cycle time of the machine may be lengthened).

So while the functionality of these complex microcoded instructions (e.g. string moves, decimal arithmetic) is still necessary, a RISC-based system is confronted with a classic space-time dilemma: if the compilers are given sole responsibility for generating the necessary sequences, the resulting in-line code expansion becomes a problem; but if procedure calls to library routines are used for each operation, the overhead expense incurred (i.e. parameter passing, stack usage, etc.) is unacceptable.

In an effort to retain the advantages associated with each approach, the alternative concept of "millicode" was developed. Millicode is PA-RISC's simulation of complex microcoded instructions, accomplished through the creation of assembly-level subroutines that perform the desired tasks. While these subroutines perform comparably to their microcoded counterparts, they are architecturally similar to any other standard library routines, differing only in the manner in which they are accessed. As a result, millicode is portable across the entire family of PA-RISC machines, rather than being unique to a single machine (as is usually the case with traditional microcode).

There are many advantages to implementing complex functionality in millicode, most notably cost reduction and increased flexibility. Because millicode routines reside in system space like other library routines, the addition of millicode has no hardware cost, and consequently no direct influence on system cost. It is relatively easy and inexpensive to upgrade or modify millicode, and it can be continually improved in the future. Eventually, it may be possible for individual users to create their own millicode routines to fit specific needs.

Because it is costly to architect many variations of an instruction, most fixed instruction sets contain complex instructions that are overly general. Examples of this are the MVB (move bytes) and MOVE (move words) instructions on the HP3000, which are capable of moving any number of items from any arbitrary source location to any target location. Although the desired functionality is achieved with such generalized complex instruments, the code that is produced often lacks the optimization that could have been achieved if all information available at compile time had been utilized. On microcoded machines, this information (concerning operands, alignment, etc.) is lost after code generation and must be recreated by the microcode during each execution; but on PA-RISC machines, the code generators can apply such information to select a specialized millicode routine that will produce a faster run-time execution of the operation than would be possible using a generalized routine. For example, the move routines can execute much faster if they can assume a specified alignment, and therefore eliminate any error checking of that type.

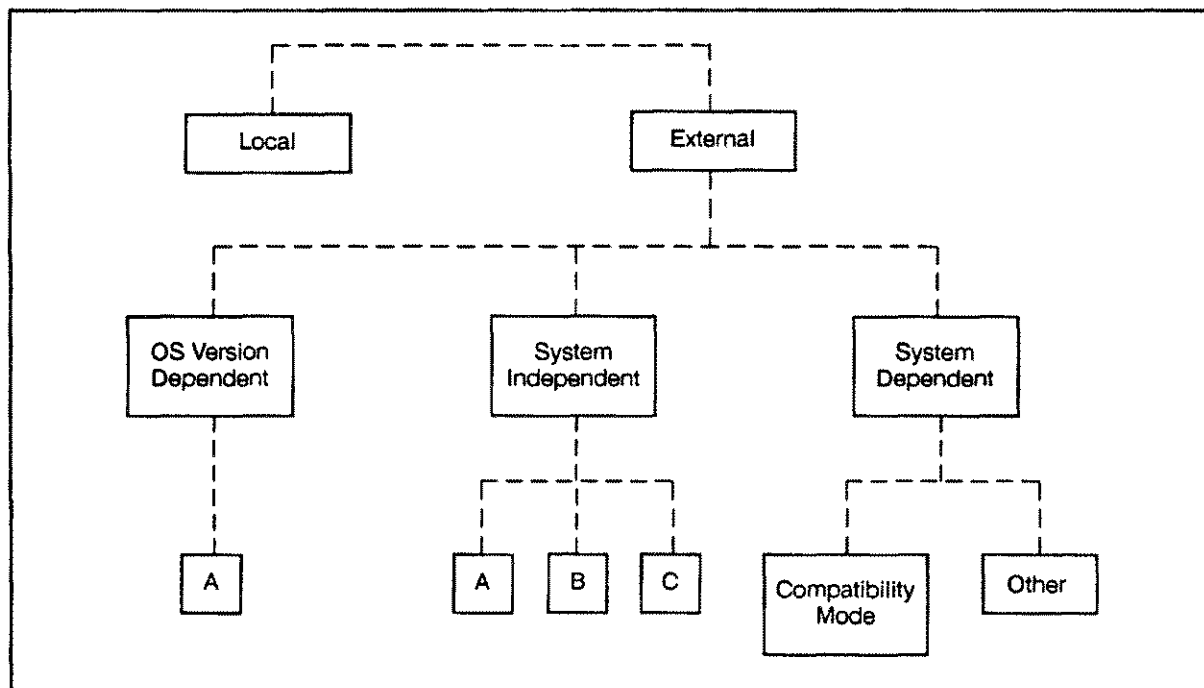
The size and number of millicode routines are not constrained by the architecture or hardware considerations. This is because millicode resides in code libraries that can be managed in the same way as other run-time libraries. A consequence of not being bound by restrictive space considerations is that compilers can be developed with many more specialized functions in millicode than would be possible in a microcoded architecture, and thus are able to create more optimal solutions for specific source code occurrences.

Millicode routines are accessed through a mechanism similar to a procedure call, but with several significant differences. In general terms, the millicode calling convention stresses simplicity and speed, utilizing registers for all temporary argument storage and eliminating the need for the creation of excess stack frames. Thus, a great majority of the overhead expense associated with a standard procedure call is avoided, thereby reducing the cost of execution. (However, there are exceptions to these conventions, which are discussed in more detail throughout this chapter.)

The guidelines for the inclusion of a routine in the millicode library are not completely determined, but the general considerations are frequency of usage, processor expense (number of cycles necessary for execution), and size. Most routines perform common, specific tasks (such as integer multiply or divide), and require very little or no memory access.

## 6.1. The Millicode Hierarchy

In an effort to define and classify the various types of millicode, Figure 6-1 shows a conceptual schematic layout of the existing millicode “family”. The labels in the boxes are briefly described following the diagram, and will be discussed in greater detail throughout the remainder of the chapter.



LG200010\_020

Figure 6-1. Millicode Overview.

### 6-2 Millicode Calls

---

## 6.2. Descriptions

*Local:* The millicode routines contained in the executable code of each process requiring them (i.e. not shared). Similar to local library routines, but with the faster access advantage.

*External:* True, shared millicode, residing in system space.

*System Dependent:* Millicode which is useful to only one operating system; can be used at the discretion of the operating system.

*System Independent:* Millicode which is intended to run on more than one operating system (i.e., routines required by language code generators).

*OS Version Dependent:* The same as System Independent but programs using this millicode may only run on the same version of the OS on which it was linked. (i.e. it may not be portable.) Useful on MPE XL, for system libraries which come with a specific OS.

*Compatibility Mode:* System dependent code (on MPE XL) which assists both the emulator and translator.

*System Dependent - Other:* Additional system dependent routines.

*A, B, C:* Classes of millicode that differ by location in virtual space and accessibility. The classifications are made on the basis of performance and size considerations.

---

## 6.3. Introduction to Local and External Millicode

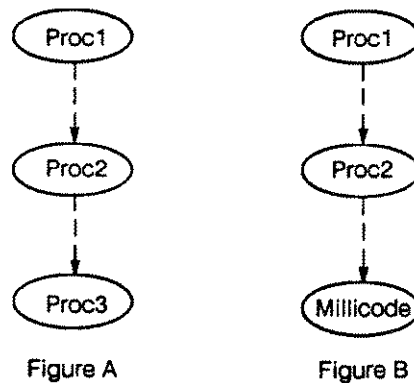
As pictured in Figure 6-1, millicode is generally divided into two main categories: local and external. Although it appears that the two types are co-existing, this is NOT the case; at present all millicode is local, whereas in the future it is possible that the great majority, if not all, of millicode will be external.

The two types of millicode are easily differentiated by the way in which they are accessed and used: Local millicode is linked with and executed by any process that requires it, while external millicode is handled in much the same manner as a standard shared library (i.e. one system-resident copy that is shared by all processes as is necessary).

## 6.4. Efficiency Factors

There are several conditions that contribute to the increased efficiency of millicode calls. The primary one is the fact that any standard routine that makes only millicode calls is considered to be a leaf routine, thereby eliminating the overhead expense (i.e. frame allocation) that would have been added by the presence of calls to standard procedures. A higher percentage of leaf routines improves overall efficiency, since a standard procedure call is much more costly than a millicode call. (in terms of stack frame allocation and usage, etc.). This and other major factors contributing to this efficiency are summarized below:

1. The compiler is able to identify whether a routine is a leaf or not; it only builds a complete stack frame for non-leaf routines. In the diagram below, a stack frame is created upon the call to Proc1 in both Fig. A and Fig. B. In Fig. A, another frame is then allocated for Proc2 when the compiler realizes that Proc2 will be subsequently calling Proc3, whereas in Fig. B, no additional frame is necessary because the compiler realizes that Proc2 is only making a millicode call.



LG200010\_021

2. More parameters can be passed in registers to a millicode routine than to a standard procedure call.
3. The compiler knows more about millicode routines at compile time than it does about user-defined procedures, so it can perform some intra-procedural optimizations across the millicode call.
4. The millicode calling mechanism is often faster than the standard procedure call; a millicode routine is called through a branch (BL or BLE) directly to the routine or to a pointer to the routine.

---

## 6.5. Making a Millicode Call

A call to a millicode routine can only be made from the assembly level. It is currently not possible to directly call a millicode function from high-level programming languages.

It is intended that the standard register usage conventions be followed, with two exceptions:

1. The return address (MRP) is passed in gr31; and
2. Function results are returned in gr29.

There are, however, many non-standard practices regarding millicode register usage.

Local millicode can be accessed with three different methods, depending on its location relative to currently executing code. These three methods are:

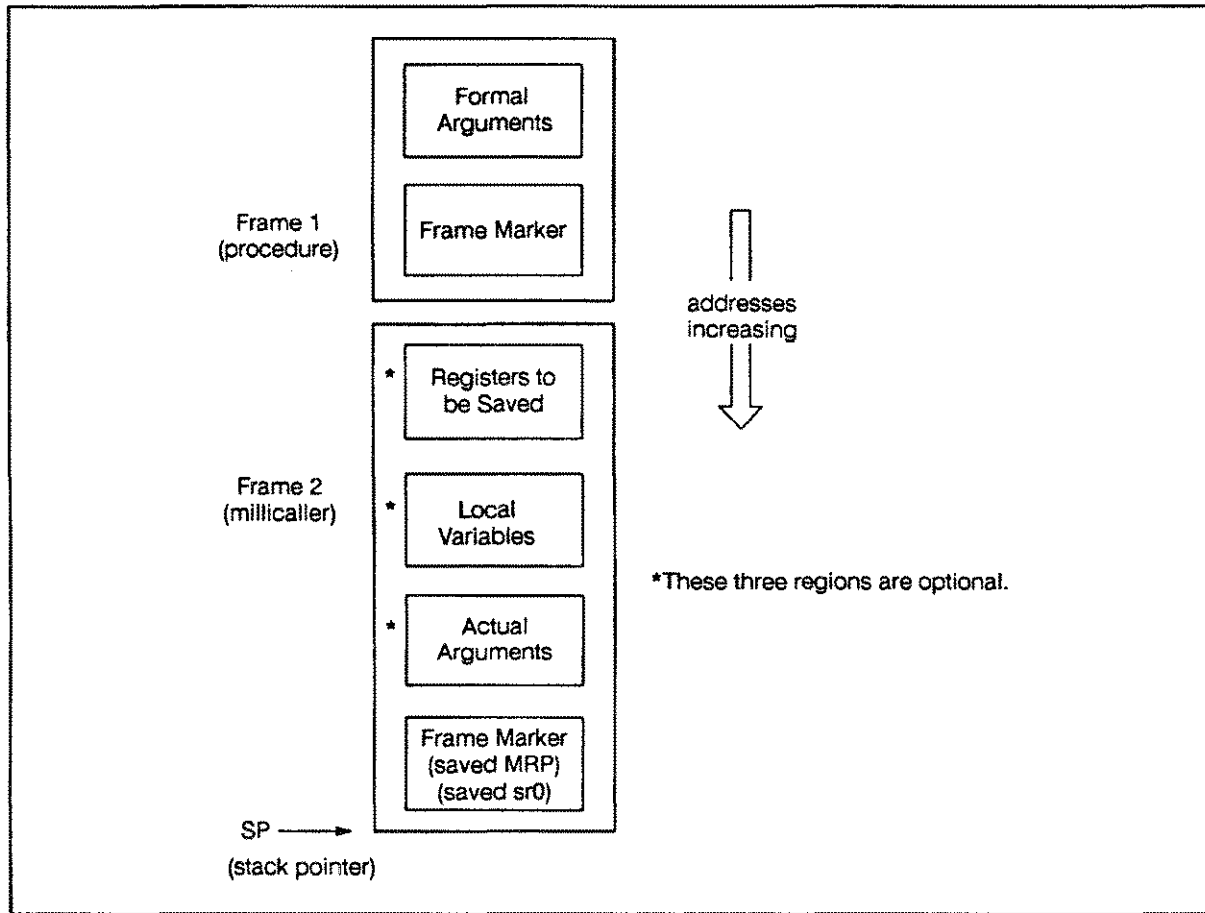
1. A standard Branch and Link (BL), if the millicode is within 256K bytes of the caller,
2. A BLE instruction, if the millicode is within 256K bytes of a predefined code base register, and
3. The two-instruction sequence (LDIL,BLE) that can reach any address or a BL with a linker-generated stub.

External millicode is accessed by method 2 above. (e.g. it can use sr6 or sr7 with a base register of 0.)

---

## 6.6. Nested Millicode Calls

Millicode routines may call other millicode routines, but (at present) cannot call other standard user-defined routines. In order for nested millicode calls to occur, however, the millicaller must allocate a stack frame and save the MRP in the current RP word (SP-20) and sr0 in the static link word (SP-16) of the frame marker area of the new frame. The layout of a frame generated for a nested millicode call is shown in Figure 6-2.



LG200010\_022

Figure 6-2. Millicode Stack Storage Layout.



## Stack Unwinding

---

This chapter is primarily intended for assembly language programmers. Some additional detail is provided for application developers who may have a need to use the stack unwind library routines.

Stack unwinding refers to the processes of procedure trace-back and context restoration, both of which have several possible system and user-level applications. A software stack unwinding convention is necessary on PA-RISC because in the event of an interruption of execution, there is insufficient information directly available to perform a comprehensive stack trace. The stack trace is the basic operation performed in context restoration.

Some important tools are heavily dependent on the presence of the stack unwinding facility. For example, system dump analysis tools examine all system processes that were running at the time of a system crash, an operation which involves multiple stack traces. Symbolic debuggers require the ability to display the state of the call stack at any point during a program's execution. Many language-specific features such as the *ESCAPE* mechanism in HP Pascal also require stack unwinding capabilities.

The information necessary to perform a stack trace, such as the size of each frame, is not available on the stack. One approach to supporting stack unwind would have been to generate information at run time and store it in the frame marker. This would have required extra instructions to be executed for most procedure calls. The designers of the PA-RISC calling convention chose not to use this approach under the assumption that stack unwinding is infrequently necessary.

Alternatively, stack unwind information is generated once at compile time and stored in a static data structure called the *unwind table*. An unwind table is automatically built into each program file by the linker.

Each entry in the unwind table contains two addresses which describe a region of code, typically the starting and ending address of a procedure. Each entry also contains an *unwind descriptor* which holds information about the frame and register usage of that region. When an unwind operation is required, the unwind table is searched to find the region containing the instruction where the exception or interrupt occurred.

The implementation details of stack unwinding are unimportant for most high-level programs and programmers. Assembly programmers, however, need to follow the documented guidelines outlined in this chapter to ensure that programs can be successfully unwound.

Further details are available in Appendix C and D. Appendix C explains the functionality of the PA-RISC unwind library routines. These can be used by programmers who want or need to access stack unwind functionality from their applications. Appendix D provides a more detailed description of what is involved in implementing customized stack unwind routines.

---

## 7.1. Requirements for Successful Stack Unwinding

Unwind depends crucially on the ability to determine, for any given instruction, the state of the stack and whether that instruction is part of a procedure entry or exit sequence. In particular, instructions that modify SP or RP must be made known to the unwind routines. Furthermore, it is necessary that all the callee-saves registers be saved at the dedicated locations on the stack (which are documented later in this chapter).

To guarantee that a routine is unwindable, the assembly programmer should strictly adhere to the stack and register usage conventions described in this manual. It is mandatory that the procedure entry and exit sequences conform to the standard specifications. All procedures generated by HP's compilers will automatically meet all these requirements and hence will be unwindable.

The assembler provides several directives that help in making routines completely unwindable. The `.ENTER` and `.LEAVE` directives will automatically generate the standard entry and exit sequences. The code sequences generated by these directives are determined by the options specified in the `.CALLINFO` directive. In rare cases, it may be necessary to generate non-standard stack frames or to create multiple unwind regions for the same routine. These cases can be handled with proper use of the `.CALLINFO`, `.ENTRY`, `.EXIT`, `.PROC` and `.PROCEND` directives as documented in the *PA-RISC Assembly Language Reference Manual*. (An example is given in section 7.6.)

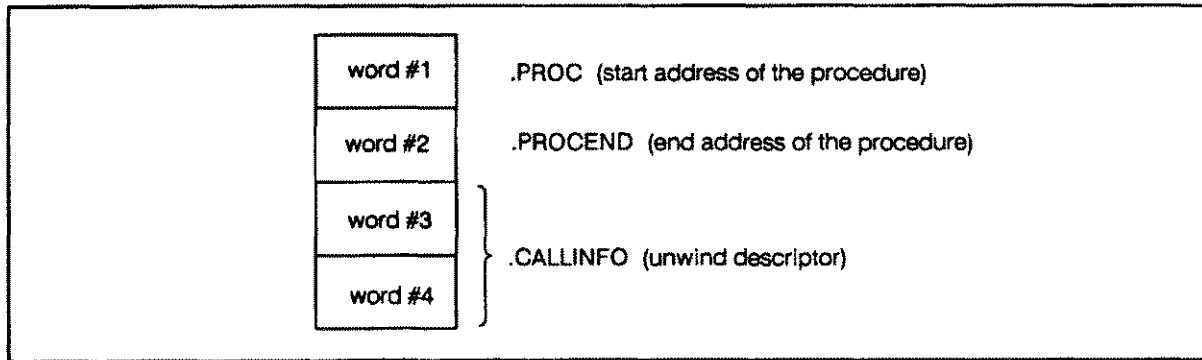
To successfully perform a stack trace from any given instruction in a program, the following requirements must be met:

- The specified instruction must lie within a standard code sequence, as specified above.
- Caller-save registers must be saved and restored across a call (if their contents are live across a call).
- Unwind table entries must be generated for each routine, and for any discontinuous regions of code.
- The frame size for each routine must be the same as is stated in the unwind descriptor for that routine.
- The use of RP (or MRP) in each routine must conform to the specifications stated in the unwind descriptor for that routine.

The minimum requirements for a successful context restoration are:

- All requirements for a stack trace (as above) must be met.
- The use of the callee-saves registers in each routine must conform to the specifications given in the unwind descriptor for that routine.

The assembler generates fixup requests for the linker based on the information made available to it by the programmer in the various procedure entry, exit, and call directives. The linker builds the unwind descriptors based on these fixup requests. The unwind descriptors describe the stack and register usage information for a particular address range and the length of the entry and exit sequences. The unwind descriptors are four word entities with the following format:



LG200010\_023

The linker sorts all the unwind descriptors according to the address range they refer to and places them in a separate subspace. Most stack unwind functions depend on the unwind entries being sorted properly.

---

## 7.2. Unwinding From Millicode

The one type of standard call from which unwindability cannot be guaranteed is the millicode call. This is because the assembler cannot automatically generate the standard entry and exit sequences for millicode routines that allocate additional stack space. Fortunately, relatively few millicode routines require the creation of a stack frame. It is possible, however, to support unwinding from such routines (i.e., nested millicode calls), provided that the millicode routine which allocates the stack space is written so that it uses the correct entry and exit sequences. It is the responsibility of the author of the specific routine to incorporate these provisions into the actual code.

---

## 7.3. Instances in Which Unwinding May Fail

A successful stack trace may not be possible in the following situations:

- Procedures that have multiple (secondary) entry points.
- Code sequences in which DP (gr27) is modified. Note that DP should never be altered by user code, only by system code as is absolutely necessary.

---

## 7.4. Callee-Saves Register Spill

For a procedure to be unwindable, the callee-saves registers must be stored in the correct location within the stack frame. The registers will be stored in the correct locations when the standard entry and exit sequences generated by the `.ENTER` and `.LEAVE` are used. The stack unwinding utilities may fail if an interrupt occurs on an instruction in a non-standard entry or exit sequence. For this reason, it is advisable that assembly programmers use `.ENTER` and `.LEAVE` rather than create their own entry and exit sequences.

If you don't use the `.ENTER` and `.LEAVE` directives, then callee-saves registers should be saved within the procedure's stack frame as follows:

Any floating-point registers are saved starting at the double-word at the bottom of the current stack frame, the address in `SP` on entry to the procedure. Register `fr12` should be stored at this location, with subsequent callee-saves registers saved in numeric order in the double-words immediately following.

Any general registers are saved starting at the first word after the last callee-saves floating-point register is saved. Register `gr3` should be stored first, with subsequent registers saved in numeric order in the words immediately following.

Callee-saves space register `sr3` is saved by moving its contents to a general register with an `MFSP` instruction and then storing it in the first double-word aligned address immediately following the last callee-saves general register. (Note that if an odd number of general registers are spilled, there will be an unused word of memory between the last general register spilled and the saved `sr3` value.)

---

## 7.5. The HP Pascal ESCAPE Mechanism

The implementation of the `TRY`, `RECOVER`, and `ESCAPE` statements in HP Pascal have system-wide implications related to stack unwinding. Besides the unwind table, HP Pascal programs will also have a *recover table* in the `$RECOVER$` subspace. This table is used to determine the execution resumption point based on the value of the program counter at the point where an `ESCAPE` is performed.

The routines which perform non-local escapes use the routines from the unwind library, described in Appendix C.

## 7.6. A Simple Example

These are examples to illustrate some of the rules to keep in mind while writing assembly code.

This example illustrates how the stack gets laid out at the entry code with the callee-saves registers. Note that the `.callinfo` requests that GR3, GR4 and FR12 ... FR14 get stored in the stack. It also allocates 32 bytes of space for local variables. The entire frame size including the frame marker is 64 bytes.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;
    .space $TEXT$
    .subspa $CODE$

stack_layout
    .PROC
    .CALLINFO      entry_gr=4, entry_fr=14, frame=32
    .EXPORT        stack_layout,code
;; If you used a .enter directive, you'd see this code automatically
;; generated by the assembler.
;;    .ENTER

    fstds,ma      fr12,8(sp)
    fstds,ma      fr13,8(sp)
    fstds,ma      fr14,8(sp)
    stwm          r3,40(sp)
    stw           r4,-36(sp)

;; Assume this is the body of the program.
    ...
    ...

;; This is the exit sequence. This would have been automatically generated
;; if you used .leave directive.
;;    .LEAVE
    ldw           -36(sp),r4
    ldwm          -40(sp),r3
    fldds,mb      -8(sp),fr14
    fldds,mb      -8(sp),fr13
    bv            r0(rp)
    fldds,mb      -8(sp),fr12
    .PROCEND
```

## 7.7. An Advanced Example: Multiple Unwind Regions

In the following example function, there is one entry point `$$foo`. A frame of 64 bytes is allocated, and there is an instruction that will cause a trap to be taken. Notice that this is a millicode routine, which uses non-standard entry/exit sequences. Please read through the assembler manual and familiarize yourself with assembler directives before reading the example.

```

;-----
;; Control does not start here. This is not the entry point to the
;; example. See below for entry point $$foo.
;; Note that callinfo, tells that this is a millicode region, and that
;; stack has gr3 stored into it.
;; First Unwind region, Stack has a 64 byte frame.
        .PROC
        .CALLINFO      millicode, entry_gr=3, frame=60
r3_zero_trap

        ldwm    -64(sp),r3

        .PROCEND

;-----
;; This not the entry point. Control falls through here from the region
;; defined above. The only change here, is that the stack has been killed.
;; This is communicated by a callinfo without any frame information.
;; Second Unwind region. Stack is empty.
        .PROC
        .CALLINFO      millicode
region_two

        be 0(sr0,r31)
        addito,=      0,r0,r0

        .EXIT
        .PROCEND

;-----
;-----MILlicode ENTRY POINT, $$foo
;; Control starts here. The callinfo tells that the stack has gr3 in it. By
;; default this means that the stack has been bumped up by 64 bytes.
        .PROC
        .CALLINFO      millicode, entry_gr=3, frame=60
        .ENTRY
$$foo

;; Store r3 on stack.
        stwm    r3,64(sp)
;;
;; Assume a benign body for this routine, i.e neither the stack, nor RP,
;; DP get changed.
;;
;; Trap if r3 is zero.
        comb,= 0,r3,r3_zero_trap
        nop

;; Restore r3 and exit.
        be     0(sr0,r31)
        ldwm  -64(sp),r3

        .EXIT
        .PROCEND
;-----

```

This example illustrates:

- Non-standard entry-exit sequences, notice `.ENTER` and `.LEAVE` are not used.
- A need to create multiple unwind regions to maintain unwindability.

Control enters at `$$foo`. The non-standard entry code bumps the stack up by 64 bytes and stores `gr3` in the stack. The `.CALLINFO` indicates that there is a 64 byte stack frame. If the conditional branch (`COMB`) is not taken, then `gr3` is restored from the stack and the stack is decremented properly. Note that in this region the stack always has a frame. If the condition succeeds, then we branch to `r3_zero_trap`. This region is physically above the entry point and is also marked as a region with a 64 byte frame. The stack is decremented in this region, and control flows into the third region. Note that in this region, the stack will be empty. To indicate this change to the state of the stack, another `.CALLINFO` directive is used. When the `ADDITO` instruction traps, the trap handler can successfully complete a stack trace or resume execution in the context of a procedure somewhere below the current stack pointer.





## Standard Procedure Calls

---

The assembly listing on the following pages was produced by the Pascal compiler (without optimization) when given the source code shown below. The approximately equivalent ANSI C and FORTRAN versions of the source code are also shown. Significant differences are noted (either in the source code or the documentation of the assembly code) where appropriate.

---

### A.1. Pascal Source Code

```
program test;

function mul (a,b : integer): integer;
begin
    mul := a * b;
end;

procedure proca (    a,b : integer;
                   VAR c,d : integer;
                   e,f : integer);

begin
    c := a + b;
    d := mul(a,b);
end;

procedure one;
var a,b,c,d,e,f : integer;
begin
    a := 5;
    b := 10;
    proca (a,b,c,d,e,f);
    e := c + d;
    f := e;
end;

begin
    one;
end.
```

---

## A.2. ANSI C Source Code

```
#include <stdio.h>

int mul (int a, int b)

{
    return a * b;
}

void proca (int a, int b, int *c, int *d, int e, int f)
{
    *c = a + b;
    *d = mul(a, b);
}

void one (void)
{
    int a, b, c, d, e, f;
    a = 5;
    b = 10;
    proca (a, b, &c, &d, e, f);
    e = c + d;
    f = e;
}

main (void)
{
    one();
}
```

---

## A.3. FORTRAN Source Code

```
call one
end

subroutine one
integer*4 a,b,c,d,e,f
a = 5
b = 10
call proca(a,b,c,d,e,f) --> Note: In FORTRAN, all parameters are passed
                                by reference, so it is impossible to
                                simulate the difference between
                                Pascal's VAR and Value parameters.

e = c + d
f = e
return
end

subroutine proca(a,b,c,d,e,f)
integer*4 a,b,c,d,e,f
c = a + b
d = mul(a,b)
return
end

function mul(a,b)
integer*4 a,b
mul = a*b
return
end
```

## A.4. Assembly Listing

The numbers and letters in parenthesis are used as labels for the documentation that follows the listing.

```
.SPACE $TEXT$                (a)
.SUBSPA $CODE$,QUAD=0,ALIGN=8,ACCESS=44,CODE_ONLY

mul
    .PROC                      (b)
    .CALLINFO CALLER,FRAME=-8,ENTRY_SR=1
    .ENTRY
;    .EXPORTED

    LDD    40(30),30           ; (11)
    STW    26,-76(0,30)       ;
    STW    25,-80(0,30)       ;
    LDW    -76(0,30),26       ;
    .CALL ; in=25,26; out=29;
    BL     $$muloI,31         ;
    LDW    -80(0,30),25       ;

    STW    29,-40(0,30)       ; (12)
    LDW    -40(0,30),28       ;
00002711 (dummy label)
    BV     0(2)                ; (13)
    .EXIT                          ; (c)
    LDD    -40(30),30         ;
    .PROCEND;in=25,26;out=28;

proca
    .PROC                      (b)
    .CALLINFO CALLER,FRAME=0,ENTRY_SR=1,SAVE_RP
    .ENTRY
;    .EXPORTED

    STW    2,-20(0,30)        ; (7)
    LDD    48(30),30          ;

    STW    26,-84(0,30)       ; (8)
    STW    25,-88(0,30)       ;
    STW    24,-92(0,30)       ;
    STW    23,-96(0,30)       ;
    LDW    -84(0,30),1        ; (9)
    LDW    -88(0,30),31       ;
    ADDD   1,31,19            ;
    LDW    -92(0,30),20       ;
    STW    19,0(0,20)         ;

    LDW    -84(0,30),26       ; (10)
    .CALL ; in=25,26; out=28;
    BL     mul,2              ;
    LDW    -88(0,30),25       ;

    LDW    -96(0,30),21       ; (14)
    STW    28,0(0,21)         ;
00002712 (dummy label)
    LDW    -68(0,30),2        ; (15)
    BV     0(2)                ;
    .EXIT                          ; (c)
    LDD    -48(30),30         ;
    .PROCEND; in=23,24,25,26;
```

```

one
      .PROC (b)
      .CALLINFO CALLER,FRAME=32,SAVE_RP
      .ENTRY
;      .EXPORTED

      STW 2,-20(0,30) ; (3)
      LDD 80(30),30 ;

      LDI 5,22 ; (4)
      STW 22,-60(0,30) ;
      LDI 10,1 ;
      STW 1,-64(0,30) ;

      LDW -60(0,30),26 ; (5)
      LDW -64(0,30),25 ;
      LDD -68(30),24 ;
      LDD -72(30),23 ;
      LDW -76(0,30),31 ;
      LDW -80(0,30),19 ;
      STW 31,-52(0,30) ;
      STW 19,-56(0,30) ;

      .CALL ; in=23,24,25,26; (6)
      BL proca,2 ;
      NOP ;

      LDW -76(0,30),20 ; (16)
      LDW -80(0,30),21 ;
      ADDD 20,21,22 ;
      STW 22,-76(0,30) ;
      LDW -76(0,30),1 ;
      STW 1,-80(0,30) ;

00002713 (dummy label)
      LDW -100(0,30),2 ; (17)
      BV 0(2) ;
      .EXIT ; (c)
      LDD -80(30),30 ;
      .PROCEND ;

```

```

PROGRAM
_start
    .PROC                (c)
    .CALLINFO CALLER,FRAME=0,SAVE_SP,SAVE_RP
    .ENTRY
;    .EXPORTED

    STW    2,-20(0,30)    ;    (1)
    LDO    48(30),30      ;
    STW    0,-4(0,30)    ;
        :
        :
    < calls to system process initialization procedures >
    < these would not appear in the C compiler output >
        :
        :
    .CALL                ;    (2)
    BL     one,2          ;
    NOP                    ;
        :
        :
    < calls to system process termination procedures >
    < these would not appear in the C compiler output >
        :
        :
    LDW    -68(0,30),2    ;    (18)
    BV     0(2)           ;
    .EXIT                ;    (c)
    LDO    -48(30),30     ;
    .PROCEND ; in=24,25,26;

    .SPACE $TEXT$                (a)
    .SUBSPA $CODE$

    .EXPORT mul,ARGWO=GR,ARGW1=GR,RTNVAL=GR (d)
        :
        :
    < .EXPORT list continues >

    < .IMPORT list begins >      (e)
        :
    .IMPORT $$mul0I,MILLICODE
    .END

```

---

## A.5. Code Description

The relevant assembler directives are summarized in Appendix B, and the other compiler-generated information is briefly explained following the code comments.

(Numbers below correspond to those accompanying the blocks of code; they appear in the order in which they would be executed. In other words, the code documentation follows the program flow of control. The code listed in section A.4 does not adhere to the requirement that stack frames be multiples of 64 bytes in size.)

1. The beginning of the main program block (note that the main program is handled in much the same manner as a standard procedure). Because other procedures will be subsequently called, it is necessary to store the Return Address and allocate a stack frame. The Return Pointer (RP), which is currently in gr2, is first stored onto the stack at SP-20, and then SP (gr30) is incremented (by 64 bytes) in order to create the new frame. A zero value is stored in the previous SP field of the frame marker in order to signify the termination point for stack unwinding. (In the compiled C code, this initialization would not appear because the outer block is handled differently.)
2. CALL to the procedure *one*. The return pointer (RP), which is the address of the second instruction following the BL, is put into gr2. The delay slot (i.e. the instruction following the branch) is NOP because there is no operation that the compiler could have inserted there.
3. ENTRY to procedure *one*. Again, this is a non-leaf procedure, so it is necessary to store RP onto the stack at SP-20 and then allocate a new frame by incrementing SP (this time the increment is 64 bytes in order to accommodate the local variables).
4. The immediate values 5 and 10 are loaded into gr22 and gr1 respectively, and these registers are stored onto the stack at SP-60 and SP-64. This block correspond to statements *a:=5* and *b:=10*.
5. Loading arguments (into caller-saves registers). This can be divided into three categories. First, the values stored on the stack at SP-60 and SP-64 (corresponding to *a* and *b*) are loaded into arg0 and arg1 (gr26 and gr25). Second, the addresses SP-68 and SP-72 are loaded into arg2 and arg3 (gr24 and gr23). Corresponding to variables *c* and *d*, these two arguments are loaded with addresses rather than actual values due to the fact that they are being passed by reference (i.e. VAR parameters). Third, the values stored on the stack at SP-76 and SP-80 (corresponding to *e* and *f*) are loaded into gr31 and gr19 respectively (these two are serving as scratch registers), and then stored onto the stack at SP-52 and SP-56. Note that these two parameters must be stored onto the stack because the argument registers have already been filled. (In the compiled FORTRAN code, it would become evident that all parameters are passed by reference, as in the Second category above, as is dictated by the FORTRAN language.)
6. CALL to procedure *proca*. Note that the .CALL directive is followed by a note indicating that arguments will be passed to the procedure in gr23-26. The delay slot is filled with a NOP, although it could have been filled with another operation (e.g. one of the preceding STW or LDW instructions). As with all BL instructions, the return address is simultaneously loaded into gr2 (or gr31 for millicode).
7. ENTRY to procedure *proca*. As before, this is a non-leaf procedure, so it is necessary to store RP at (SP-20) and allocate an additional stack frame by incrementing SP (64 bytes in this case).

8. The values held in the four argument registers (gr26-23) are stored onto the stack in the fixed arguments area of the PREVIOUS (caller's) frame. This is determined by subtracting the size of the current frame (48 bytes) from the offset (84, 88,...), and using the result as the offset into the previous frame. These words correspond to the parameters *a* through *d*. (Note that these Store operations are actually unnecessary, and would probably be removed by the optimizer.)
9. The words at SP-84 and SP-88 (parameters *a* and *b*) are loaded into gr1 and gr31 respectively and the add operation ( $a+b$ ) is performed, with the result being put into gr19. After SP-92 (which contains the address of *c*) is loaded into gr20, the gr19 value is stored at that address.
10. CALL to function *mul*. After the two parameters (*a* and *b*) are loaded into arg0 and arg1 (gr26 and gr25), the branch is made to *mul*, and the return address is put into gr2. Note that in this case, the delay slot is filled with an operation (the loading of the *b* value).
11. ENTRY to function *mul* and CALL to millicode routine *muloI*. Although frame is allocated because the function return value will later be temporarily stored onto the stack, but RP is not stored onto the stack because no additional procedure calls will be made. (This temporary frame is actually unnecessary, and would be removed by the optimizer.) The two arguments (*a* and *b*, in arg0 and arg1) are stored onto the stack, and then reloaded into registers in order to be sent to the millicode routine that will perform the multiply operation. Then the branch is made to the millicode routine ( $$$muloI$ ), with the return address being stored in gr31 (MRP).

This code could be further optimized by accessing the arguments directly from the registers in which they enter *mul*, thereby eliminating the argument stores and loads.

12. The millicode return value (in gr29) is stored onto the stack, and subsequently loaded into gr28, which is the procedure return register (ret0). This sequence would probably be optimized to be a simple *COPY 29,28* instruction.
13. EXIT from function *mul*. Deallocate the local frame, and return back to the caller (*proca*). The BV (Branch Vectored) instruction, which also uses gr2 as the return pointer, accomplishes this return.
14. RETURN from *mul* to *proca*. The value stored at SP-96 (the address of *d*) is loaded into gr21, and then the return value (in gr28) is stored at that address.
15. EXIT from procedure *proca*. The return address is loaded into gr2 from the *RP* field of the Previous frame, and the branch is made to that address. The delay slot is filled with the instruction that deallocates the local frame by decrementing SP.
16. RETURN from *proca* to *one*. The values stored at SP-68 and SP-72 (the current values of *c* and *d*) are loaded into gr20 and gr21, and the add operation ( $c+d$ ) is performed, with the result being put into gr22. This result is then stored onto the stack at SP-76, which is the location assigned to *e*. Finally, the value stored in the *e* word is reloaded (into gr1), and then stored into SP-80, which is the location of *f*. (This is the  $f:=e$  operation.)
17. EXIT from *one*. The return address is taken from its memory location (SP-100) and loaded into gr2, the local frame is deallocated by decrementing SP, and the branch is taken to the return point in the main program.
18. EXIT from main program. The return address (i.e. to the system) is loaded into gr2, the local frame is deallocated by decrementing SP, and the branch is made to the system address.



---

## A.6. Other Compiler-Generated Information

(Letters correspond to those accompanying directive blocks.)

- a. `.SPACE` and `.SUBSPA` specify the proper space and subspace for the instructions that follow.
- b. This four-directive sequence appears at the beginning of every procedure. The directives are summarized in Appendix B.
- c. The directives `.EXIT` and `.PROCEND` appear at the end of every procedure. Their functions are summarized in Appendix B.
- d. The `.EXPORT` list, which is the list of all procedures contained within this process that can be globally accessed.
- e. The `.IMPORT` list, which is the list of all procedures that this process is dependent upon (includes the system initialization and process termination procedures mentioned in the main program code).

---

## A.7. External Calls

The assembly code on the page after next was produced by the MPE XL Pascal compiler from the Pascal source code shown on the next page. A few additional notes concerning the code sample:

In the source code, an external call situation has been simulated by assuming that the callee (*one*) resides in a different load module than the caller (*two*).

The assembly listing has been abbreviated to include only the code associated directly with the source code. In the complete listing, there would be calling and called stubs for all calls to process initialization and termination procedures (which occur in the outer block/main program as noted below) preceding the section of code shown here.

Because the use of the `CALLX` millicode is transparent to the user, it has been just referenced as being *in system space* to avoid all of the excess detail that would be necessary to use actual addressing. A similar liberty has been taken in a few other cases; where actual offsets appear in the code, they have been eliminated to achieve simplicity.

The code sample is accompanied (in the left margin) by arrows that follow the flow of control. These arrows function exactly as those used in the flow diagrams in the text; in this case, the starting point is in the main program block, near the bottom of the code sample. Furthermore, all *critical points* have been labeled with numbers and documented on the page following the assembly code (just as was done in the local call example).

The assembly code is lightly documented; the code used in the stubs is documented in detail in Chapter 5, Import and Export Stubs, and the small amount of code present other than in stubs is basically the same as that used in any local procedure call.

## A.7.1. Pascal Source Code

```
program extcall;
:
:
    (* procedure one is in module 2; therefore an external call
       is necessary in order for the call from two to be successful *)

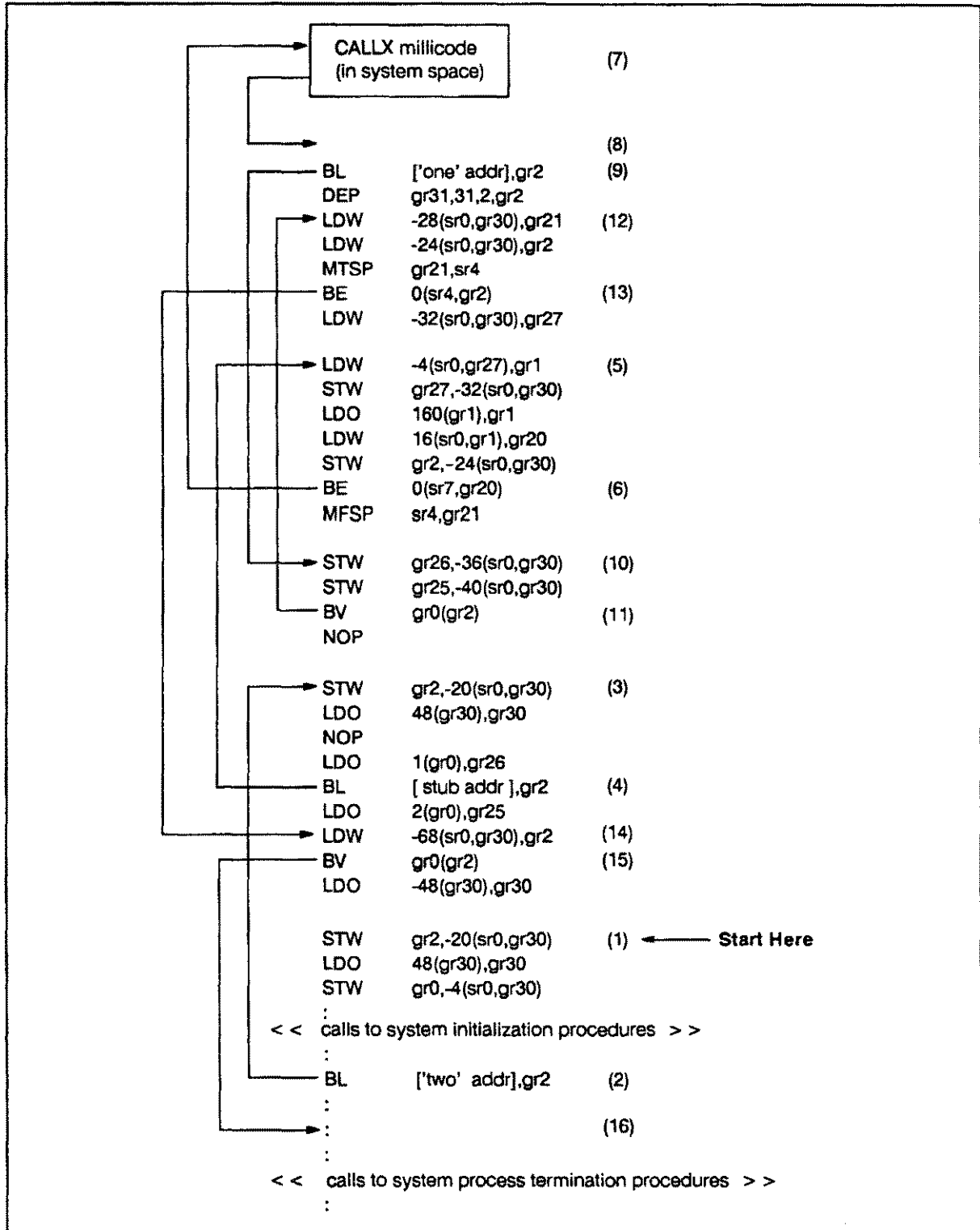
procedure one (a,b : integer);
begin
end;
:
:
    (* procedure two is in module 1, i.e. the same module
       as the main program block. Bar calls one *)

procedure two;
begin
    one (1,2);
end;

begin
    two;
end.
```

## A.7.2. Assembly Code

The numbers in parenthesis follow the flow of control and are used as labels for the documentation that follows.



LG200010\_030

---

## A.8. Assembly Documentation

1. Start of outer block/main program. RP is saved, the stack frame is allocated, and the unwind delimiter is initialized.
2. Call to procedure *two*. This is a local call, so the branch goes directly to the procedure, with no stub interaction.
3. Entry to procedure *two*. RP is saved, the frame is allocated, and the constant values 1 and 2 are loaded into argument registers 1 and 2 (gr26 and gr25).
4. Branch to Calling Stub. As far as the procedure *two* is concerned, this is the call to the procedure *one*, but the branch actually goes to the calling stub that is necessary because this is an external procedure call.
5. Entry to Calling Stub attached to *two*. The calling stub performs as is documented in detail in Chapter 5, Calling Stub, of the text.
6. Call to CALLX millicode. The Branch External instruction is used in order to reach the CALLX millicode routine.
7. Execution of CALLX millicode; it performs exactly as is documented in Chapter 5, External Procedure Call Millicode (CALLX), of the text, and then branches to the called stub that is attached to procedure *one*.
8. Entry to called stub attached to procedure *one*.
9. Branch to procedure *one*. The standard Branch and Link instruction is used to reach the actual code for the external procedure *one*.
10. Entry to procedure *one*. The arguments in gr26 and gr25 are stored onto the stack (this is not necessary, and only remains because the code has not been optimized).
11. Exit from *one* / Branch back to called stub. The standard return instruction (BV) is used here, although the branch is actually going to the called stub, and not directly to the caller.
12. Re-entry to called stub. The remainder of the called stub performs as is documented in detail in Chapter 5, Called Stub, of the text.
13. Exit from called stub / branch back to procedure *two*. The Branch External instruction is used to reach to actual code for the procedure *two* (the caller).
14. Return to procedure *two* from called stub. The previous RP is loaded into gr2 from the stack, and the frame is deallocated.
15. Exit from procedure *two* / branch back to main program block. The standard procedure return is made, because this is a local return.
16. Return to main program block from procedure *two*. After return, calls are made to the system process termination procedures, and then the frame is deallocated and the return is made to the system.

## Summary of PA-RISC Assembler Procedure Control

---

The following table summarizes the PA-RISC assembler directives that are used to control procedure calling:

Directive	Function
.CALL	Specifies that the next statement is a procedure call.
.CALLINFO	Provides information necessary for generating Entry and Exit code sequences and for creating unwind descriptors.
.ENTRY	Marks the entry point of the procedure/unwind region. (when .ENTER is not used)
.EXIT	Marks the return point of the procedure/unwind region. (when .LEAVE is not used)
.ENTER	Marks the entry point of the procedure being called; causes the assembler to produce the entry code sequence.
.LEAVE	Marks the exit point of the procedure being called; causes the assembler to produce the exit code sequence.
.PROC	Marks the first statement in a procedure.
.PROCEND	Marks the last statement in a procedure.



## The Stack Unwind Library

---

This appendix explains the data structures and functionality in the unwind library. This information can be useful in further understanding the behavior of the unwinding process, as well as in writing applications that require stack unwinding functionality.

### C.1. The Unwind Descriptor

When the assembler sees procedure directives such as `.ENTER` or `.LEAVE`, it builds fixup requests for the linker. Using the information in these fixup requests, the linker builds a 4-word unwind descriptor for each unwind region. These descriptors monitor a particular code address range, typically an entire procedure. The unwind descriptors provide information about the stack size, registers usage, and the lengths of the entry and exit sequences. The linker sorts these entries in the increasing order of code addresses and places them in a separate subspace.

Here is a C language declaration of the unwind descriptor.

```

struct unwind_table_entry {
    unsigned int region_start;           /* Word 1 */
    unsigned int region_end;            /* Word 2 */
    unsigned int Cannot_unwind:1;       /* Word 3 */
    unsigned int Millicode:1;
    unsigned int Millicode_save_sr0:1;
    unsigned int Region_description:2;
    unsigned int reserved1:1;
    unsigned int Entry_SR:1;
    unsigned int Entry_FR:4;
    unsigned int Entry_GR:5;
    unsigned int Args_stored:1;
    unsigned int Variable_Frame:1;
    unsigned int Separate_Package_Body:1;
    unsigned int Frame_Extension_Millicode:1;
    unsigned int Stack_Overflow_Check:1;
    unsigned int Two_Instruction_SP_Increment:1;
    unsigned int Ada_Region:1;
    unsigned int reserved2:4;
    unsigned int Save_SP:1;
    unsigned int Save_RP:1;
    unsigned int Save_MRP_in_frame:1;
    unsigned int reserved3:1;
    unsigned int Cleanup_defined:1;
    unsigned int MPE_IL_interrupt_marker:1; /* Word 4 */
    unsigned int HP_UX_interrupt_marker:1;
    unsigned int Large_frame_r3:1;
    unsigned int reserved4:2;
    unsigned int Total_frame_size:27;
};

```

Each of the fields in this structure are used as follows:

- [1] *Cannot\_unwind*: One if this region does not follow unwind conventions and is therefore not unwindable; zero otherwise. (Creation of non-unwindable assembly code is strongly discouraged.)
- [2] *Millicode*: One if this region is a millicode routine; zero otherwise.
- [3] *Millicode\_save\_sr0*: One if this (millicode) routine saves sr0 in its frame (at current\_SP - 16); zero otherwise.
- [4] *Region\_description*: Describes the code between the starting and ending offsets of this region:
  - 00: Normal (entry point at start of region, exit point at end; contains no other entry/exit points)
  - 01: Entry point only (contains no exit point)
  - 10: Exit point only (contains no entry point)
  - 11: Discontinuous (contains no entry or exit point)

Normal context is code that falls between the last entry point and first exit point of a routine.

Entry point only context is code that makes up an alternate entry point. It consists of entry code inserted by the assembler or compiler as well as user code. It does not contain exit code.

Exit point only context is code that makes up an alternate exit point. It consists of exit code inserted by the assembler or compiler as well as user code. It does not contain entry code.

Discontinuous context is code within an assembled or compiled routine that is either not preceded by some entry point or not followed by some exit point.

One unwind table entry is generated per routine, plus one for each additional entry point, exit point, and discontinuous region. Normally, all unwind descriptors are identical except for the *Region\_description* field. The entry and exit points to any region are marked using the `.ENTRY` and `.EXIT` assembler directives.
- [5] *Entry\_SR*: One if the sole entry-save space register sr3 is saved/restored by the associated entry/exit code sequence; zero otherwise.
- [6] *Entry\_FR*: The number of entry-save floating-point registers saved/restored by the associated entry/exit code sequence.
- [7] *Entry\_GR*: The number of entry-save general registers saved/restored by the associated entry/exit code sequence. Note that the semantics of this field are different from those of the similarly named field of the `.CALLINFO` directive to the assembler. For example, a value of 5 in this field would mean that gr3 through gr7 (inclusive) have been saved in the entry save code.
- [8] *Args\_stored*: One if this region's prologue includes storing any arguments to the routine in memory in the architected locations; zero otherwise. (Note: this bit may not be correct if the associated routine was compiled with optimization, as the optimizer may remove initial stores of arguments, but will never clear this bit.)



- [9] *Variable\_Frame*: Indicates that this region's frame may be expanded during the region's execution (using the Ada dynamic frame facility). Such frames require different unwinding techniques.
- [10] *Separate\_Package\_Body*: Indicates the associated region is an Ada separate package body. It has no frame of its own, but uses space in a parent frame to save RP and spill any entry save registers.
- [11] *Frame\_Extension\_Millicode*: Indicates the associated region is a special millicode routine which implements the Ada frame extension operation.
- [12] *Stack\_Overflow\_Check*: Indicates the associated region has an Ada stack overflow check in its entry sequence(s).
- [13] *Two\_Instruction\_SP\_Increment*: Indicates the associated (Ada) region had a large frame such that two instructions were necessary to produce that portion of the frame increment which cannot be deduced from the frame size field in the unwind descriptor.
- [14] *Ada\_Region*: One if the associated region should be treated as an Ada routine. Currently this bit is intended for use by the Ada exception handler.
- [15] *Save\_SP*: One if the entry value of SP is saved by this region's entry sequence in the current frame marker (current\_SP - 4); zero otherwise.
- [16] *Save\_RP*: For non-millicode, one if the entry value of RP is saved by the entry sequence in the previous frame (at previous\_SP - 20); zero otherwise. For millicode, one if the entry values of MRP and sr0 are saved by the entry sequence in the current frame (at current\_SP - 20 and current\_SP - 16, respectively); zero otherwise. If this bit is one, the Save\_MRP\_in\_frame and Millicode\_save\_sr0 bits are ignored.
- [17] *Save\_MRP\_in\_frame*: One if the entry value of MRP is saved by the entry code in the current frame (at current\_SP - 20); zero otherwise. Applies only to millicode.
- [18] *Cleanup\_defined*: The interpretation of this field is dependent upon the language processor which compiled the routine.
- [19] *MPE\_XL\_interrupt\_marker*: One if the frame layout corresponds to that of an MPE XL interrupt marker.
- [20] *HP\_UX\_interrupt\_marker*: One if the frame layout corresponds to that of an HP-UX interrupt marker.
- [21] *Large\_frame\_r3*: One if gr3 is changed during the entry sequence to contain the address of the base of the (new) frame.
- [22] *Total frame size*: The amount of space, in 8-byte units, added to SP by the entry sequence of this region. This space includes register save and spill areas, as well as padding. This quantity is needed during unwinding to locate the entry-save register save area. It is also used to determine the value of previous\_SP if it was not saved in the stack marker.

---

## C.2. Unwind Utility Routines

Unwind utility routines provide the core functionality of the stack unwind mechanism. They can be used to create powerful unwind applications. These routines are present and have identical functionality on all PA-RISC systems running HP-UX or MPE XL.

### 1. U\_get\_unwind\_table

```
struct utable {
    unsigned unwind_table_start;
    unsigned unwind_table_end;
};
struct utable U_get_unwind_table(unsigned int dp_value);
```

Returns the code offsets of the start and end of the unwind table of a given object module. The unwind table is word-aligned. It takes the DP value for the object module where the unwind table is stored. It returns the offset of the start of the unwind table, and the offset of the first word beyond the unwind table.

This is the interface for assembly programmers:

ARG0: DP value of routine being unwound to.  
RET0: Offset (in space of routine being unwound to) of start of unwind table.  
RET1: Offset (in space of routine being unwound to) of first word beyond end of unwind table.

### 2. U\_get\_unwind\_entry

```
int U_get_unwind_entry(
    unsigned int PC;
    unsigned int Space_id;
    unsigned int table_start;
    unsigned int table_end);
```

Given the *PC\_offset* value of interest and the start and end of the associated unwind table, returns the code offset (in *PC\_space*) of the associated unwind table entry. If no unwind table entry exists, -1 is returned. Typically the *table\_start* and *table\_end* is found using the *U\_get\_unwind\_table* routine.

This is the interface for assembly programmers:

ARG0: PC value to look up.  
ARG1: Space id of table.  
ARG2: Offset of start of unwind table.  
ARG3: Offset of first word beyond end of unwind table.  
RET0: Offset of unwind table entry associated with PC value;  
-1 if none exists.

This routine requires that the unwind table is sorted in increasing order of starting addresses. It does a binary search of the table to get to the entry corresponding to the input PC value.

### 3. U\_get\_previous\_frame

```
struct current_frame_def {
    unsigned cur_fsize; /* Frame size of current routine. */
    unsigned cursp; /* The current value of stack pointer. */
    unsigned currls; /* PC-space of the calling routine. */
    unsigned currlo; /* PC-offset of the calling routine. */
    unsigned curdp; /* Data Pointer of the current routine. */
    unsigned toprp; /* Initial value of RP. */
    unsigned topmrp; /* Initial value of MRP. */
    unsigned topsr0; /* Initial value of sr0. */
    unsigned topsr4; /* Initial value of sr4. */
    unsigned r3; /* Initial value of gr3. */
    unsigned cur_r19; /* GR19 value of the calling routine. */
    /* Used only in HP-UX. */
};

struct previous_frame_def {
    unsigned prev_fsize; /* frame size of calling routine. */
    unsigned prevSP; /* SP of calling routine. */
    unsigned prevRLS; /* PC_space of calling routine's caller. */
    unsigned prevRLO; /* PC_offset of calling routine's caller. */
    unsigned prevDP; /* DP of calling routine. */
    unsigned udescr0; /* low word of calling routine's unwind */
    /* descriptor. */
    unsigned udescr1; /* high word of calling routine's unwind */
    /* descriptor. */
    unsigned ustart; /* start of the unwind region. */
    unsigned uw_index; /* index into the unwind table. */
    unsigned uend; /* end of the unwind region. */
    unsigned prev_r19; /* GR19 value of the caller's caller. */
};

int U_get_previous_frame(
    struct current_frame_def *curr_frame;
    struct previous_frame_def *prev_frame);
```

Given a *PC\_space*, a *PC\_offset* value that is a return link to the caller, the frame size, and the DP and SP values of the called routine, these routines return the frame size, the DP and SP values of the caller's frame, and the (*PC\_space*, *PC\_offset*) value that is a return link to the caller's caller.

The routine returns:

- 0 normal;
  - 1 if curRLS, curRLO is nil, indicating stack was fully unwound;
  - 4 if error occurs during linker stub unwinding  
other negative values less than -1 may be used in the future to indicate additional unexpected (internal) errors.
- positive if the frame is not unwindable for some reason.

Values currently used:

- 1 - no unwind\_descriptor
- 0x7fffffff - cannot\_unwind bit on in previous unwind descriptor

This is the interface for assembly programmers:

- ARG0: Pointer to an eleven-word area of memory that contains the current frame info.
- ARG1: Pointer to an eleven-word area of memory defined on exit as per definition of the *previous\_frame\_info* structure.
- RET0: Return value defined on exit.

This routine is designed to enable access to the previous frame on the stack with input information about the current state. You may call this iteratively by setting the *cur* fields to the appropriate machine state, and then copying the first five *prev* values (and the *prev\_r19 value*) into the corresponding fields for successive calls, until end-of-stack is reached.

When a nonzero value is returned, the fields that would normally get defined on exit are undefined.

If the frame of the called routine is the topmost frame on the stack when unwinding commences, *cur\_frsz* should be zero on the initial call.

---

## C.3. Recover Utility Routines

These routines can be used to resume execution at a specific point if something unexpected happens. The HP Pascal run-time libraries use these routines to recover from traps and to execute non-local ESCAPE statements.

### C.3.1. The Recover Table

Entries are sorted by ascending address. Each entry is three words long. Its format is as follows:

```
struct recover_table_entry {
    unsigned TRY_start;    /* Starting offset (from sr4) of TRY region.*/
    unsigned TRY_end;     /* Ending offset (from sr4) of the
                           instruction following TRY region.    */
    unsigned RECOVER_start; /* RECOVER block offset for associated TRY
                           region (execution resumes here).    */
};
```

#### 1. U\_get\_recover\_table

```
struct rtable {
    unsigned recover_table_start;
    unsigned recover_table_end;
};
struct rtable U_get_recover_table(unsigned int dp_value);
```

Returns the code offsets of the start and end of the recover table of a given object module. The recover table is word-aligned. It takes the DP value for the object module where the recover table is stored. It returns the offset of the start of the recover table, and the offset of the first word beyond the recover table.

This is the interface for assembly programmers:

ARG0:	DP value of routine associated with PC value of interest.
RET0:	Offset (in space of routine being unwound to) of start of recover table.
RET1:	Offset (in space of routine being unwound to) of first word beyond end of recover table.

## 2. U\_get\_recover\_address

Given the *PC\_offset* value of interest and the location of the associated recover table, returns the code offset (in *PC\_space*) of the associated recover block. If the *PC\_offset* does not point to a try block, -1 is returned.

```
int U_get_recover_address(  
    unsigned int PC;  
    unsigned int Space_id;  
    unsigned int rtable_start;  
    unsigned int rtable_end);
```

This is the interface for assembly programmers:

- ARG0: *PC\_offset* to look up.
  
- ARG1: Offset (in space of routine being unwound to) of start of recover table.
  
- ARG2: Offset (in space of routine being unwound to) of first word beyond end of recover table.
  
- ARG3: Space id of recover table.
  
- RET0: Recover address with actual execution level, or -1 if not found.

---

## C.4. Obtaining a Stack Trace

Applications can obtain stack traces easily using the following utility routine:

```
U_STACK_TRACE()
```

This routine can be called from any place without any arguments. It will print the stack trace from the caller's frame onwards onto the standard output stream.

---

## C.5. Errors From The Unwind Library

You may get the following error messages while using the Unwind routines in the Unwind library available on your system. A brief explanation follows each error message, describing the cause of the message and any preventive measures.

1. Escape executed outside of a Try block; CODE = <escape-num> (UNWIND 9)

This tells you that a Pascal non-local escape was performed outside a TRY block. Hence the unwind process could not find a corresponding RECOVER block.

Check your code to see if there is a RECOVER block for the TRY within which this escape is being made. If this is not the problem, then check to see if the recover tables have the proper descriptors.

2. Non-unwindable descriptor during Escape; CODE = <escape-num> (UNWIND 11).  
Non-unwindable descriptor during non-local GOTO (UNWIND 20).

During the process of unwinding, a *PC\_offset* value was encountered that did not have a proper unwind descriptor. Check the unwind descriptor for the current PC value. Most of the time, the region may be marked as non-unwindable.

3. Missing unwind descriptor during Escape; CODE = (UNWIND 10).  
Missing unwind descriptor during non-local GOTO (UNWIND 21).

Could not find an unwind descriptor in the Unwind tables for this *PC\_offset* value. Check the .CALLINFO for this region.

4. End-of-stack during non-local GOTO (UNWIND 19).

The non-local GOTO performed has an address that is not within the call chain, hence the unwinding process reached the bottom of the call stack without reaching the target of the non-local GOTO.

This problem can occur with secondary entry points, which generate multiple unwind regions. The call chain must be in the same region as the target of the GOTO.





## The Stack Unwind Process

This appendix explains the stack unwind process in detail. It is intended for users who need to develop unwind routines that go beyond what the unwind library already can do. It is assumed that the reader understands the material presented in Chapter 7 and Appendix C.

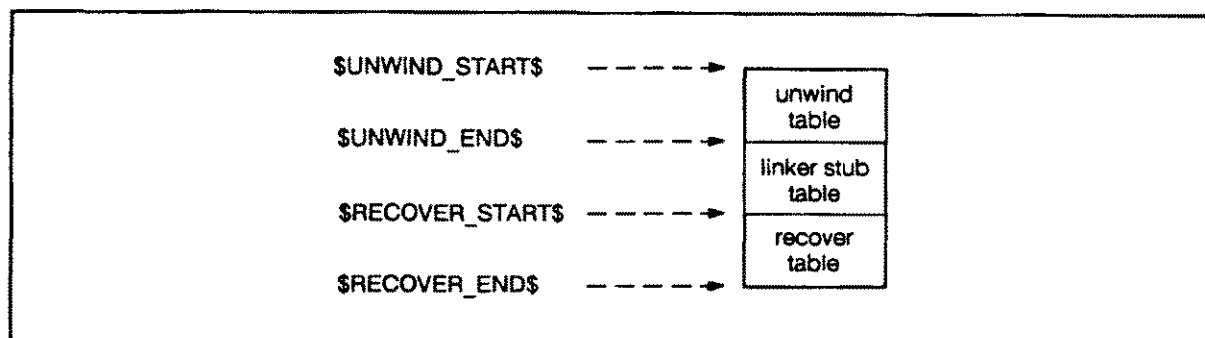
### D.1. Role of Stubs in Unwinding

Stubs are generated by the linker when a procedure makes an external call. Although there are various kinds of stubs, all of them save some data about the current location and then branch to some other location. Since it will be necessary to unwind from stubs, it is necessary to describe these regions in the unwind table. To do this, the linker generates two-word unwind descriptors for stubs. The stub-unwind descriptors have the following format:

```
word 1:  address of the first instruction of the stub
word 2:
        bits 0..3      - reserved
        bits 4..7      - type of stub
        bits 8..10     - reserved
        bits 11..15    - used only for parameter relocation stub; contains the
                        number of the instruction which stores RP on the stack
                        in the stub.
        bits 16..31    - length (# of words) of stub area
```

In some cases, a contiguous sequence of calling, called, or long branch stubs or millicode long branch can be covered by a single unwind descriptor.

The UNWIND and RECOVER subspaces point to the unwind, stub, and recover tables. These tables are arranged in code space as follows:



LG200010\_029

These four labels will be initialized at run time by the loader startup routines. These tables can be accessed by the unwind library routines *U\_get\_unwind\_table()* and *U\_get\_recover\_table()*. The recover table consists of twelve-byte entries. The first two words are the PC-start and PC-end values; the last word is the address at which the recover should take place. This table is built for the HP Pascal ESCAPE mechanism, as explained in Appendix C.

---

## D.2. Unwinding From Stubs on MPE XL

Since each stub performs a different function, the unwind routines need to perform stub-specific actions.

*Calling Stubs:* None of the significant registers are modified; RP still contains the return address.

*Called Stubs:* All significant registers are on the stack. RP and DP are stored by calling stub, and SID is stored by CALLX. Their locations are:

RP : SP - 24

SID : SP - 28

DP : SP - 32

*Parameter Relocation Stubs:* It must be determined (from the unwind descriptor) if the current address is before or after the instruction which stores RP on the stack. If it is before the current address, RP still contains the return address; otherwise the return address will be stored on the stack.

---

## D.3. Unwinding From Stubs on HP-UX

A few HP-UX specific stubs have been designed to support the shared library mechanism. Calls to external routines in HP-UX will return via an export stub. The call itself will go through an import stub, similar to the calling and called stubs of Chapter 5. The import stub stores the following registers: RP (at SP - 24) and gr19 (at SP - 32).

In the HP-UX shared library implementation, gr19 plays the role of the shared library link register. It points to a shared library descriptor. This descriptor contains a pointer to the location where the unwind tables and stub tables are located. Note that each shared library has its own tables.

When unwinding through the HP-UX-export stub, the registers RP and gr19 are restored from the stack. In most normal cases, the RP that is restored will point to the caller. But in one particular case, the RP value has the address of the export stub for *\_sigreturn()*. This happens only when the stack frame has an interrupt-marker on it, which means that a trap occurred and you are unwinding from a trap handler.

Note that these stubs will not be encountered when unwinding in the absence of shared libraries. In this case, gr19 will play the normal role of a caller-save register only.

---

## D.4. Unwinding From Millicode

The unwindability cannot be guaranteed for all millicode calls because the assembler cannot automatically generate the standard entry and exit sequences for millicode routines that utilize stack space. This does not present a major problem because relatively few millicode routines create a stack frame. It is possible, however, to support unwinding from such situations (i.e., nested millicode calls), provided that a millicode routine that allocates stack space is written so that it will independently generate the correct entry and exit sequences. It is the responsibility of the author of the specific routine to incorporate these sequences into the actual code.

Also, on HP-UX, millicode resides not as a separate shared library, but as a part of each shared library. Hence one would not encounter any stubs while unwinding from millicode routines. This is also true on MPE XL when internal millicode is used. Each load module has its own copy of millicode.

Unwinding from external millicode is slightly different than code or normal millicode because `sr4` does not track PC's SID. `Sr0` and `gr31` are used to return to the caller. Both these values must be stored if nested millicode calls are done. (This is currently done for internal millicode too, since the *Save\_RP* bit is set instead of *Save\_MRP\_in\_frame*.) External millicode and the unwind descriptors for external millicode must be at an architected location. This allows unwinding a mixture of internal and external millicode.

It is likely that stack traces may also hide the call to millicode because application programmers do not see millicode as a separate entity.

---

## D.5. Unwinding Across an Interrupt Marker

When a trap occurs in the hardware, the operating system is notified. The operating system will place an interrupt marker on the stack. This marker will contain the machine state at the time of the trap. Control comes to the user trap handler via the trap library after this.

Let us look at the situation where one needs to unwind from a trap handler. This situation will arise if a trap occurs within the handler, or if the handler performs a Pascal ESCAPE to a recover block or explicitly asks for a stack trace.

The operating system places the address of a special dummy routine as the return address from the trap handler and passes control over to the user trap handler. While unwinding from the trap handler, the `RP` value will have one of the following values:

1. The address of the special dummy routine.
2. The address of an export stub.

In the first case, the unwind descriptor for this special dummy routine (*\_sigreturn()* in case of HP-UX, and *hpe\_interrupt\_marker\_stub()* in MPE) will tell us that the item on the stack is an interrupt stack marker (by means of the interrupt marker bit). In the second case, we have to unwind across the stub. In most situations, this stub will be the stub for the special routine. In such a case, unwinding across the stub will immediately lead us to the address of *\_sigreturn()* itself.

In HP-UX alone there is one situation, however, when unwinding across an export-stub leads to another export stub. Consider the situation where the trap happens and is handled in

two different user defined shared libraries. In this case, the OS calls the export stub for the handler and places the address of the export stub of `_sigreturn()` at SP-20. In this case, while unwinding from the handler, the RP value will take us to the export-stub for the handler. Now, we can take a new RP value from SP-20. However, this points to the export-stub of `_sigreturn()`. Although this is not a problem, one needs to be aware of this while writing routines that automatically unwind across a series of stubs.

The only way to recognize the interrupt stack marker is by reading the unwind descriptor of the `_sigreturn()` routine. The format of the interrupt stack marker is defined by the operating system. The format can be found in `/usr/include/sys/signal.h` in HP-UX. After recovering the machine state from the interrupt stack marker, unwinding should proceed as normal.

---

## D.6. Advanced Example

This section presents an implementation, written in C and PA-RISC assembly language, of a routine to write a stack trace into a character string.

Since a full stack trace requires access to the symbol tables in the program file, we have omitted the symbols from the output. This example illustrates the use of `U_get_previous_frame`, which is the core of the unwind library.

```

/* Start of cfile.c */
#include <stdio.h>
typedef struct current_frame_def {
    unsigned cur_fsize; /* Frame size of current routine. */
    unsigned cursp;     /* The current value of stack pointer. */
    unsigned currls;    /* PC-space of the calling routine. */
    unsigned currlo;    /* PC-offset of the calling routine. */
    unsigned curdp;     /* Data Pointer of the current routine. */
    unsigned topvp;     /* Initial value of RP. */
    unsigned topmrp;    /* Initial value of MRP. */
    unsigned topsr0;    /* Initial value of sr0. */
    unsigned topsr4;    /* Initial value of sr4. */
    unsigned r3;        /* Initial value of gr3. */
    unsigned cur_r19;   /* GR19 value of the calling routine. */
                        /* Used only in HP-UX. */
} UWREC;

main()
{
    char str[100];
    gen_stack_trace(str);
    printf("%s\n", str);
}

gen_stack_trace(outstr)
char *outstr;
{
    unsigned sp, pc, rp;
    UWREC rec1;
    int depth;

    sp = &outstr + 9;
    pc = (* (int *) (sp - 20)) & ~3;
    rp = 0;
    get_pcspc(&rec1);
    rec1.cursp = sp;
    rec1.currlo = pc;
}

```

```

rec1.toprp = rp;

for (depth = 0; rec1.currlo; depth++) {
    sprintf (outstr, "%s (%2d) 0x%x0\n", outstr, depth, rec1.currlo);
    if ( get_NextFrame (&rec1) == -1)
        return;
    }
}

/* Get information about the next frame on stack. */
int get_NextFrame (prec1)
    UWREC      *prec1;          /* sp to return */
{
    int        stat;
    UWREC      rec2;

    stat = U_get_previous_frame (prec1, &rec2);
    if (stat) {
        fprintf (stderr, "Stack_Trace: error while unwinding stack\n");
        return (-1);
    }

    prec1->currlo      = rec2.currlo      ;
    prec1->cur_fsize   = rec2.cur_fsize   ;
    prec1->currsp      = rec2.cursp      ;
    prec1->currls      = rec2.currls      ;
    prec1->curdp       = rec2.curdp       ;

    return(0);
} /* NextFrame */

/*-----
* Start of sfile.s
*/
/*
* get_pcspc is an assembly routine that fills the pcspc field of
* current frame record with the sr4 value.
*/
    .space $TEXT$
    .subspa $CODE$
    .export get_pcspc,code
get_pcspc
    .proc
    .callinfo
mfsp    sr4,r20
bv      r0(rp)
stw     r20,8(arg0)          ; store caller's PC space
    .procend
    .end

```

